

**Spring** describes **Cloud Data Flow** as a “toolkit **for** building data integration and real-time data processing pipelines”, while **Skipper is** a tool **for** discovering, and managing the lifecycle **of**, **Spring Boot** applications.

Spring Cloud Data Flow provides schemas for H2, HSQLDB, MySQL, Oracle, Postgresql, DB2 and SqlServer that will be automatically created when the server starts.

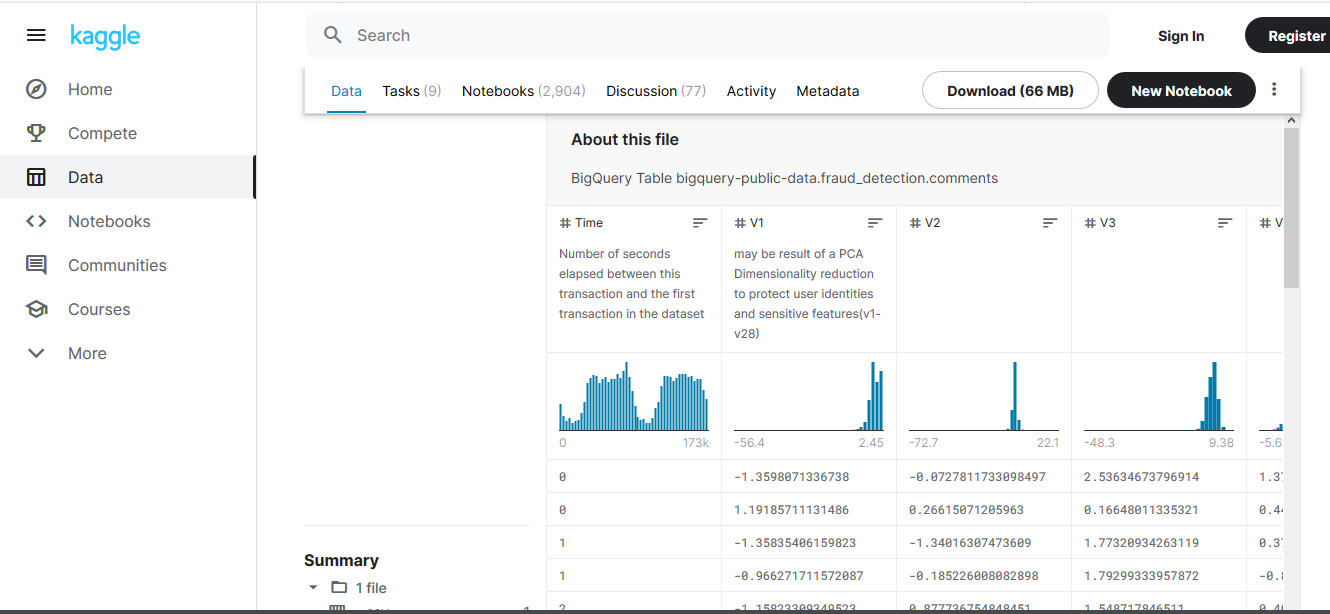
The JDBC drivers for **MySQL** (via MariaDB driver), **HSQLDB**, **PostgreSQL** along with embedded **H2** are available out of the box. If you are using any other database, then the corresponding JDBC driver jar needs to be on the classpath of the server.

The last pre-requisite is a MySql instance. If you do not have it, you will end up with an in-memory H2 database powering Data Flow. The problem with that is that you will lose all your data on a server restart. This could be actually desirable for testing, but incredibly frustrating if you invest some time in configuring your Streams only to lose them on a server restart. While creating the container, we will set a custom password and create a database for Data Flow:

**Transaction Generator:**

Standalone application that can continuously generate data records compliant with the data format specified in [Credit Card Fraud Detection]

It contains only numerical input variables which are the result of a PCA transformation. Unfortunately, due to confidentiality issues, we cannot provide the original features and more background information about the data. Features V1, V2, … V28 are the principal components obtained with PCA, the only features which have not been transformed with PCA are 'Time' and 'Amount'. Feature 'Time' contains the seconds elapsed between each transaction and the first transaction in the dataset. The feature 'Amount' is the transaction Amount, this feature can be used for example-dependant cost-senstive learning. Feature 'Class' is the response variable and it takes value 1 in case of fraud and 0 otherwise.



**Postgres DB:**

The data records are generated in random intervals and pushed into a pre-configured Database.

For confidentiality reasons the values of most of the original features, such as `V1`, `V2`, ... `V28` have been anonymized using a principal components obtained with PCA.

**Principal component analysis** (**PCA**) is a technique for reducing the dimensionality of such datasets, increasing interpretability but at the same time minimizing information loss. It does so by creating new uncorrelated variables that successively maximize variance

**CDC Source/CDC Debezium:**

**Debezium** is an open source distributed platform for change data capture. Start it up, point it at your databases, and your apps can start responding to all of the inserts, updates, and deletes that other apps commit to your databases.

The data captured from the source will then be posted to the Kafka topic. Then the processor will consume the data from Kafka for the further task.

**Fraud Detection Processor:**

Custom [Spring Cloud App Starter](https://cloud.spring.io/spring-cloud-stream-app-starters/) that extends the [**TensorFlow Processor**](https://docs.spring.io/spring-cloud-stream-app-starters/docs/Einstein.SR3/reference/htmlsingle/#spring-cloud-stream-modules-tensorflow-processor) to support a specific **TensorFlow** model build for the [Credit Card Fraud Detection](https://www.kaggle.com/currie32/predicting-fraud-with-tensorflow) solution.

**TensorFlow Processor**

A processor that evaluates a machine learning model stored in TensorFlow Protobuf format.

The TensorFlow Processor uses a TensorflowInputConverter to convert the input data into data format compliant with the TensorFlow Model used. The input converter converts the input Messages into key/value Map, where the Key corresponds to a model input placeholder and the content is org.tensorflow.DataType compliant value. The default converter implementation expects either Map payload or flat json message that can be converted into a Map.

Processor’s output uses TensorflowOutputConverter to convert the computed Tensor result into a serializable message. The default implementation uses JSON.

**Predicting Credit Card Fraud**

The goal for this analysis is to predict credit card fraud in the transactional data. I will be using tensorflow to build the predictive model, and t-SNE to visualize the dataset in two dimensions at the end of this analysis

<https://www.kaggle.com/currie32/predicting-fraud-with-tensorflow>

A pre-trained, binary model is already bundled with this project at [fraud\_detection\_graph.pb](https://github.com/tzolov/cdc-fraud-detection-demo/blob/master/fraud-detection-app-starters/spring-cloud-starter-stream-processor-fraud-detection/src/main/resources/fraud_detection_graph.pb). To (re)train the model you can use the following python script: [train-model.py](https://github.com/tzolov/cdc-fraud-detection-demo/blob/master/fraud-detection-app-starters/spring-cloud-starter-stream-processor-fraud-detection/model/train-model.py). Later is a direct copy of the [original Kaggle solution](https://www.kaggle.com/currie32/predicting-fraud-with-tensorflow), customized to export the trained model in a Protobuf fraud\_detection\_graph.pb file.

The Fraud Detection Processor expects an input JSON messages in the following format:

{

"time":453,

"v1":1.13154130503628,

"v2":-0.174361326933011,

... ,

"v28":0.034254726765817,

"amount":52.4

}

This format strictly matches the column names that types of the original dataset: <https://www.kaggle.com/mlg-ulb/creditcardfraud>

Only the Class column is intentionally removed, because the fraud/normal status of the input transaction is unknown in advance! Instead the fraud-detection processor evaluates result using the pre-trained Tensorflow model. The output result is a single json message, that can be either:

{ "detection":"FRAUD" } or { "detection":"NORMAL" }

Take in case of financial fraud detection, fraudsters are innovating faster than banks. Introducing of EMV chip did reduce card present fraud but fraudsters were quick to shift their fraud online or to card not present channel. In case of fraud detection, models might require frequent re-calibration compared to models that deal with identifying employee churn.

In data driven approach, Treat data as you treat code. Catching errors early is critical

A key aspect that data scientist needs to worry about is identifying concept drift, where data changes unpredictably over time. Check wiki link below if you need more details on concept drift

<https://en.wikipedia.org/wiki/Concept_drift>

One aspect data scientist need to understand is to differentiate drift that is anomalous with natural drifts. Natural drifts in data can be modeled using feature engineering.

Example of natural drift include Seasonality of business, Correlation of economy against employment rate or housing prices among others

Another drift is schematic or structural drift which is change due to the structure of source data.

<https://medium.com/datadriveninvestor/tensorflow-extended-tfx-data-analysis-validation-and-drift-detection-part-2-6a9c5f8c6210>

TFDV helps to catch errors early as well as identify and flag anomalous drift. Some of the key capabilities of TFDV are

* Compute and visualize summary statistics with 2 lines of code across all features
* Compares multiple datasets. Helps in identifying data and distribution skew between train, eval and serving datasets
* Automatically generates a schema based on underlying data as well as use the schema to inspect out of time datasets
* Identify anomalies, such as missing features, out-of-range values among others
* Detect data drift by looking at a series of data

[**Micrometer**](https://micrometer.io) which exposes metrics from our application, [**Prometheus**](https://prometheus.io) which stores the metric data and [**Grafana**](https://grafana.com) to visualize the data in graphs.

[**Micrometer**](https://micrometer.io) is an open-source project and provides a metric facade that exposes metric data in a vendor-neutral format that a monitoring system can understand

To be able to send custom metrics we need to import MeterRegistry from the Micrometer library and inject it into our class

It is possible to instantiate Counter from MeterRegistry

**Counter** is a single, monotonically increasing, cumulative metric.

A single metric means, that a Counter represents a single value, e.g. the number of orders created in a shop system. It’s monotonically increasing, so it can only increase, usually one-by-one. It’s a cumulative metric, so it always contains the overall value.

**Prometheus**

[Prometheus](https://prometheus.io) stores our metric data in time series in memory by periodically pulling it via HTTP. The data can be visualized by a console template language, a built-in expression browser, or by integrating [Grafana](https://grafana.com) (which we will do after setting up Prometheus).

**prometheus-rsocket-proxy** which will pull the metrics data from the micrometer and prometheus-rsocket-proxy is between micrometer and [Prometheus](https://prometheus.io).

**Grafana**

The included Prometheus browser graph is nice for basic visualization of our metrics but we will use [Grafana](https://grafana.com) instead. Grafana provides a rich UI where you create, explore and share dashboards that contain multiple graphs.

Grafana can pull data from various data sources like Prometheus, Elasticsearch, InfluxDB, etc. It also allows you to set rule-based alerts, which then can notify you over Slack, Email, Hipchat, and similar.

**InfluxDB** is in memory db which resides between [**Prometheus**](https://prometheus.io) and **Grafana**