Lab 5 Elasticsearch and Kibana (II)

In this lab, you will continue to learn Elasticsearch and Kibana, including customizing analyzer, query and aggregation, visualization, Java API for Elasticsearch and Apache Tika.

Task 1 Customize Analyzer

Recall that an analyzer in Elasticsearch is used to explain how the text will be indexed and searched. Only text fields support the analyzer mapping parameter. Analyzer can be tested by using analyzer API.

apply standard analyzer to analyze text

```
POST _analyze
{
  "text": "I REALLY like COMP810! This course is very INTERESTING :-)",
  "analyzer": "standard"
}
```

An analyzer consists of three components, i.e., character filters, tokenizer and token filters, which function in sequence.

- Character filters: Adds, removes or change characters, e.g., html_strip filter, remove all the html tags
- Tokenizer: An anlyzer contains only tokenizer, tokenizes a string, i.e., splits it into tokens.
- Token Filters: receive the output from tokenizer as input, add, remove or modify tokens. E.g., lowercase filter

let us test a customized analyzer

```
POST _analyze
{
  "text": "I REALLY like COMP810! This course is very INTERESTING :-)",
  "char_filter": [],
  "tokenizer": "standard",
  "filter": ["lowercase"]
}
```

let us test a customized analyzer by applying stemmer

```
POST /_analyze
{
  "text": "My favourite course is COMP810 interesting!",
  "char_filter": [],
```

```
  "tokenizer": "standard",
  "filter": [ "lowercase", "stemmer" ]
}
```

create analyzer_test index by adding custom analyzer

```
PUT /analyzer_test/
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_custom_analyzer": {
          "type": "custom",
          "char_filter": ["html_strip"],
          "tokenizer": "standard",
          "filter": [
            "lowercase",
            "stop",
            "stemmer"
          ]
        }
      }
    }
  }
}
```

apply my_custom_analyzer from analyzer_test index to analyze text

```
POST /analyzer_test/_analyze
{
  "analyzer": "my_custom_analyzer",
  "text": "<body>I found that COMP810 @AuT is interesting!</body>"
}
```

add a second analyzer to the analyzer_test index by updating the settings

```
PUT /analyzer_test/_settings
{
  "analysis": {
    "analyzer": {
      "my_second_analyzer": {
        "type": "custom",
        "char_filter": [
          "html_strip"
        ],
        "tokenizer": "standard",
        "filter": [
          "lowercase",
          "stop",
          "stemmer"
```

```
      ]
    }
   }
  }
}
```

apply a custom analyzer to a text field, “description” is the field name in this example

```
PUT /analyzer_test/_mapping
{
 "properties":{
  "description": {
   "type": "text",
   "analyzer": "my_custom_analyzer"
  }
 }
}
```

index a document

```
POST /analyzer_test/_doc
{
 "description": "<body>I found that COMP810 @AuT is interesting!</body>"
}
```

use the following query, you still can find the result since the same analzyer is applied to the query.

```
GET /analyzer_test/_search
{
 "query":{
  "match": {
   "description": "<div>interested</div>"
  }
 }
}
```

use below query, and cannot match any records because "that" is a stop word which is removed by the stemmer token filter of the my_custom_analyzer

```
GET /analyzer_test/_search
{
 "query":{
  "match": {
   "description": "<div>that</div>"
  }
 }
}
```

Task 2 Use Match and Term Query

Match Query returns documents that match a provided text, number, date or boolean value. The provided text is analysed before matching. The match query is the standard query for performing a full-text search, including options for fuzzy matching.

```
GET kibana_sample_data_ecommerce/_search
{
  "size": 10,
  "query": {
    "match": {
      "products.product_name": "Vest"
    }
  }
}
```

Term query returns documents that contain an exact term in a provided field. You can use the term query to find documents based on a precise value such as a price, a product ID, or a username.

Avoid using the term query for text fields. By default, Elasticsearch changes the values of text fields as part of the analysis. This can make finding exact matches for text field values difficult. To search text field values, use the match query instead.

```
GET kibana_sample_data_ecommerce/_search
{
  "size": 1,
  "query": {
    "term": {
      "_id": "oybIJ3QBnbCmhWzQfmKX"
    }
  }
}
```

```
GET kibana_sample_data_ecommerce/_search
{
  "size": 10,
  "query": {
    "term": {
      "customer_first_name.keyword": "Eddie"
    }
  }
}
```

Task 3 Use Aggregations

Statistics derived from your data are often needed when your aggregated document is large. The statistics aggregation allows you to get a min, max, sum, avg, and count of data in a single go. The statistics aggregation structure is similar to that of the other aggregations.

It is important to be familiar with the basic building blocks used to define an aggregation. The following syntax will help you to understand how it works:

```
-----
"aggs": {
   “name_of_aggregation”: {
    “type_of_aggregation”: {
      “field”: “document_field_name”
}
-----
```

- **aggs**—This keyword shows that you are using an aggregation.
- **name_of_aggregation**—This is the name of aggregation which the user defines.
- **type_of_aggregation**—This is the type of aggregation being used.
- **field**—This is the field keyword.
- **document_field_name**—This is the column name of the document being targeted.

Get number of the documents of an index

```
GET kibana_sample_data_ecommerce/_count
```

check the stats of field "total_quantity" in our data

```
GET /kibana_sample_data_ecommerce/_search
{
  "size": 0,
  "aggs": {
    "unique_skus": {
      "cardinality": {
        "field": "sku"
      }
    }
  }
}
```

Show the stats for the quantity field—min, max, avg, sum, and count values.

```
GET /kibana_sample_data_ecommerce/_search
{
  "size": 0,
  "aggs": {
    "quantity_stats": {
      "stats": {
        "field": "total_quantity"
      }
    }
  }
}
```

Filter Aggregation

As its name suggests, the filter aggregation helps you filter documents into a single bucket. Within that bucket, you can calculate metrics.

In the example below, we are filtering the documents based on the username "eddie" and calculating the average price of the products he purchased.

```
GET /kibana_sample_data_ecommerce/_search
{
  "size": 0,
  "aggs": {
    "User_based_filter": {
      "filter": {
        "term": {
          "user": "eddie"
        }
      },
      "aggs": {
        "avg_price": {
          "avg": {
            "field": "products.price"
          }
        }
      }
    }
  }
}
```

Terms Aggregation

The terms aggregation generates buckets by field values. Once you select a field, it will generate buckets for each of the values and place all of the records separately.

In our example, we have run the terms aggregation on the field "user" which holds the name of users. In return, we have buckets for each user, each with their document counts.

```
GET /kibana_sample_data_ecommerce/_search
{
  "size": 0,
  "aggs": {
    "Terms_Aggregation": {
      "terms": {
        "field": "user"
      }
    }
  }
}
```

Nested Aggregation

This is the one of the most important types of bucket aggregations. A nested aggregation allows you to aggregate a field with nested documents—a field that has multiple sub-fields.

The field type must be “‘nested’” in the index mapping if you are intending to apply a nested aggregation to it.

The sample e-commerce data which we have used up until this point hasn’t had a field with the type “nested.” We have created a new index with the field “Employee” which has its field type as “nested.”

Run the code below in DevTools to create a new index “nested_aggregation” and set the mapping as “nested” for the field “Employee.”

```
GET nested_aggregation/_search

PUT nested_aggregation
{
  "mappings": {
    "properties": {
      "Employee": {
        "type": "nested",
        "properties": {
          "first": {
            "type": "text"
          },
          "last": {
            "type": "text"
          },
          "salary": {
```

```
          "type": "double"
        }
       }
      }
     }
    }
  }
}
```

Insert a document, having an Employee array.

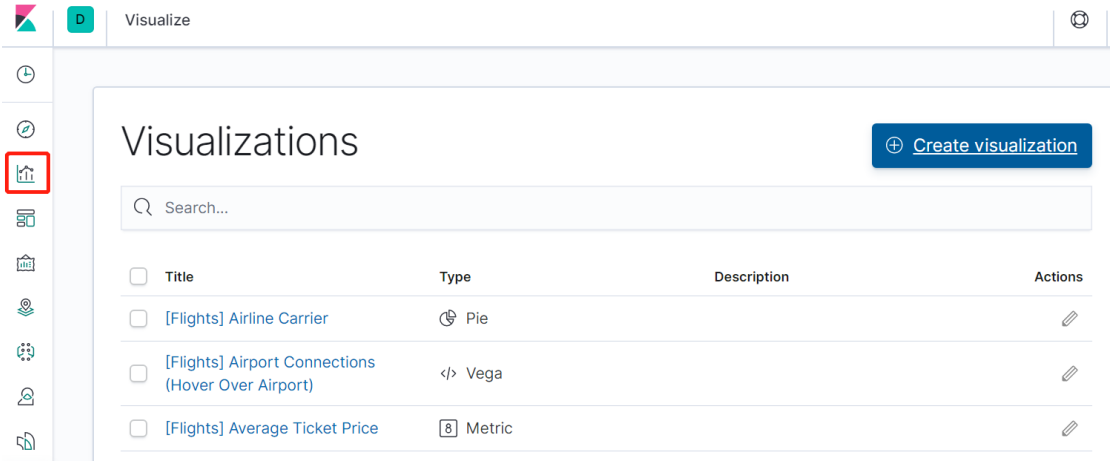
```
PUT nested_aggregation/_doc/1
{
  "group": "Logz",
  "Employee": [
    {
      "first": "Ana",
      "last": "Roy",
      "salary": "70000"
    },
    {
      "first": "Jospeh",
      "last": "Lein",
      "salary": "64000"
    },
    {
      "first": "Chris",
      "last": "Gayle",
      "salary": "82000"
    },
    {
      "first": "Brendon",
      "last": "Maculum",
      "salary": "58000"
    },
    {
      "first": "Vinod",
      "last": "Kambli",
      "salary": "63000"
    },
    {
      "first": "DJ",
      "last": "Bravo",
      "salary": "71000"
    },
    {
      "first": "Jaques",
      "last": "Kallis",
      "salary": "75000"
    }
  ]
}
```

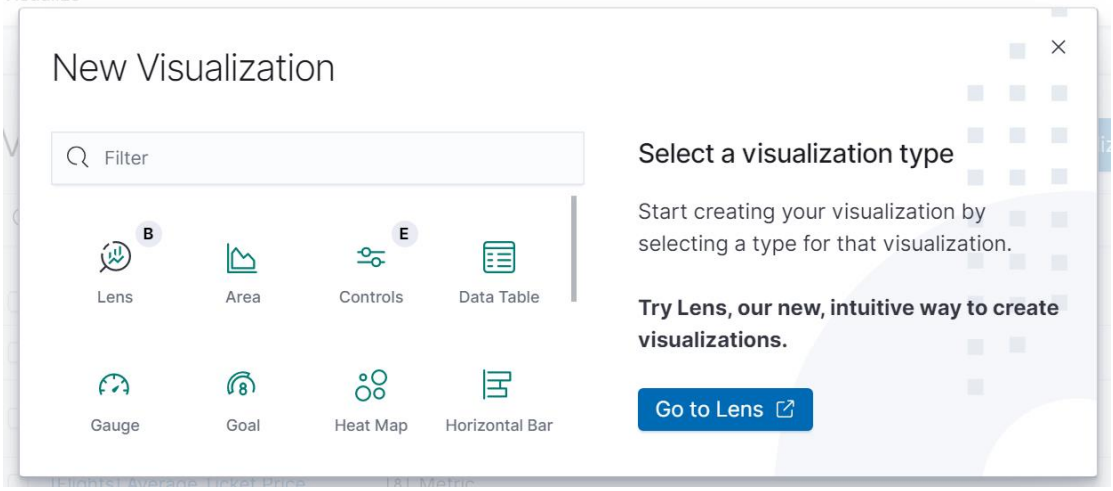
Search using nested aggregation

```
GET /nested_aggregation/_search
{
  "aggs": {
    "Nested_Aggregation": {
      "nested": {
        "path": "Employee"
      },
      "aggs": {
        "Min_Salary": {
          "min": {
            "field": "Employee.salary"
          }
        }
      }
    }
  }
}
```

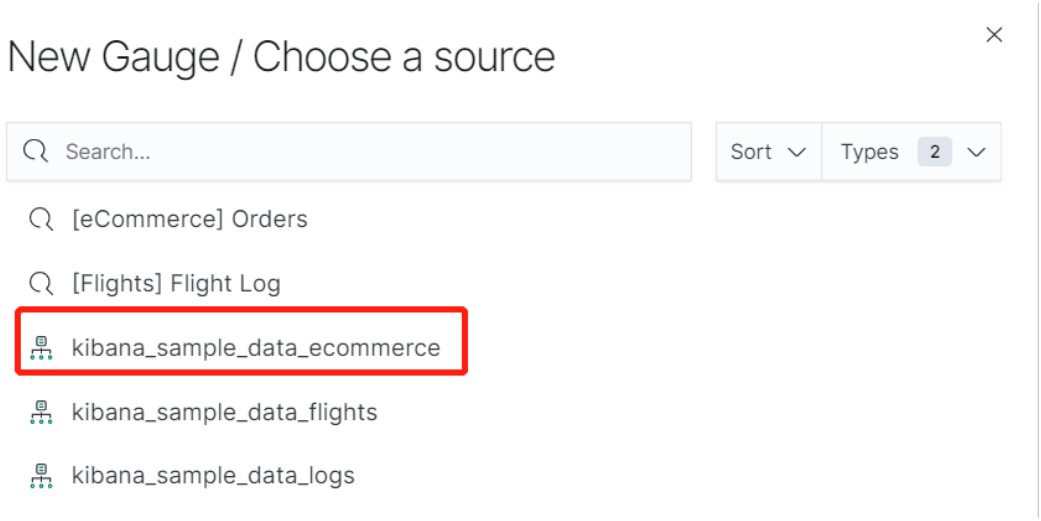
Task 4 Visualization using Kibana

Select the visualization icon, and a number of existing charts coming with the sample sets are listed. Click create visualization button.

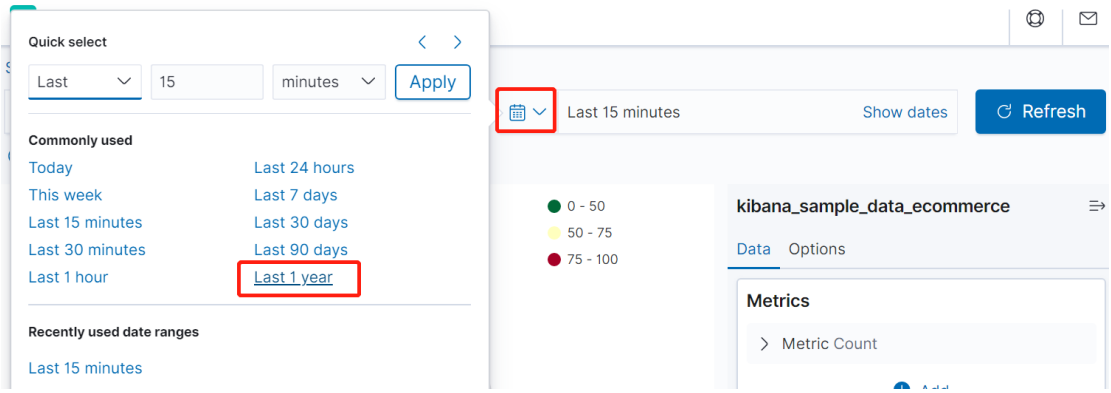




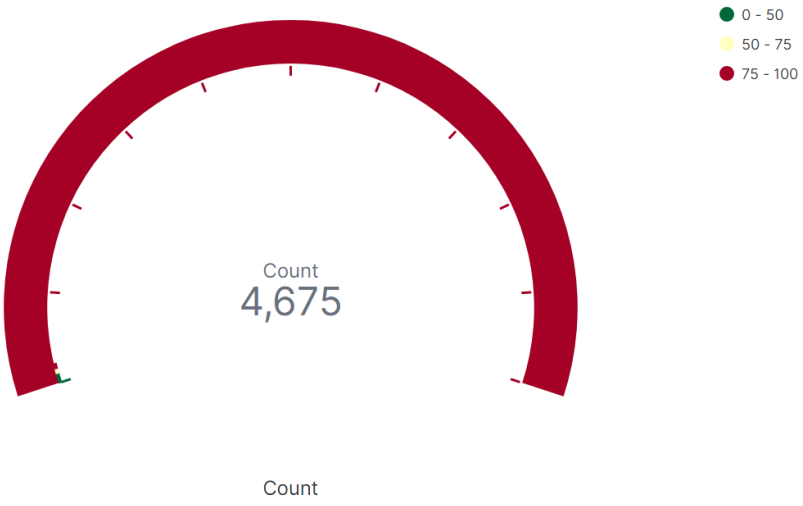
Let’s create a simple Gauge figure by choosing kibana_sample_data_ecommerce.



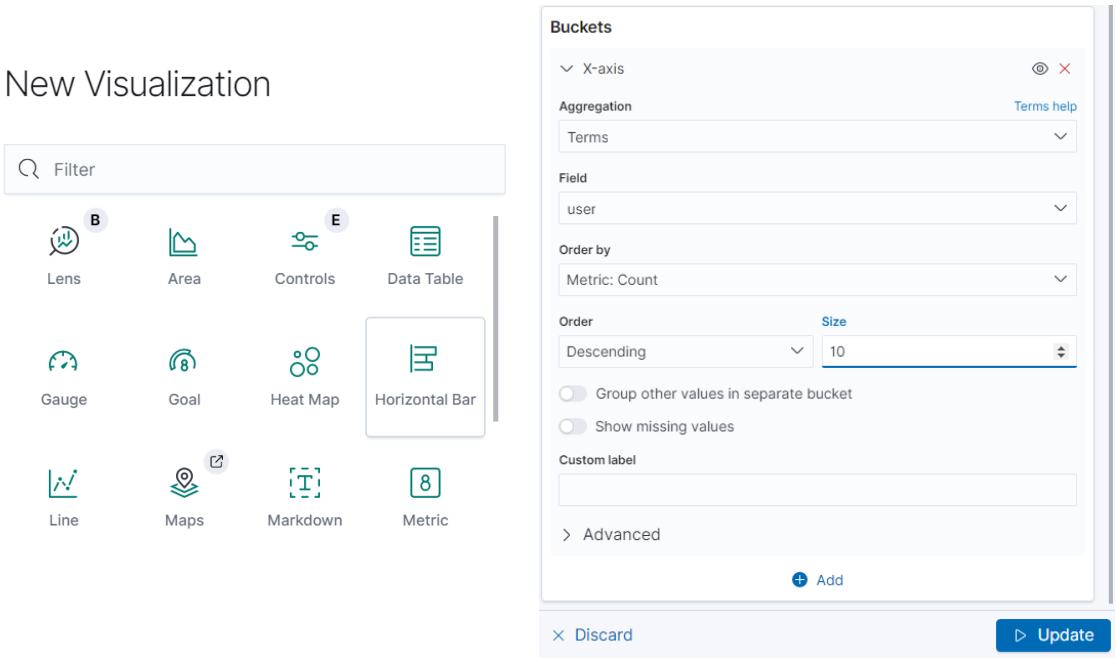
Change the date range to “Last 2 year”.



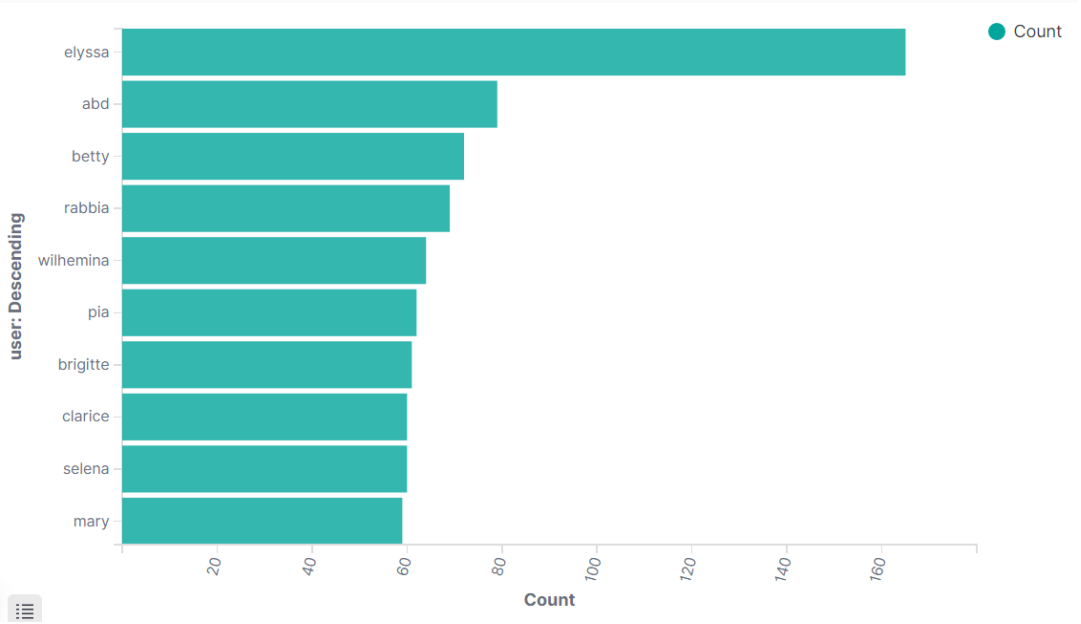
We can get the count as below.



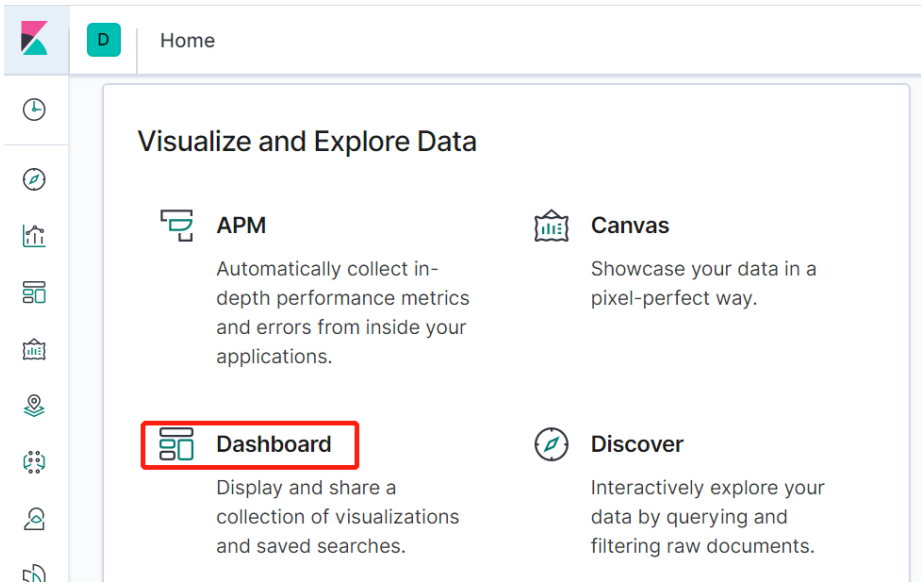
Next, create a Horizontal Bar using the e-commerce dataset. Add buckets, x-axis, and use the following options and click update.



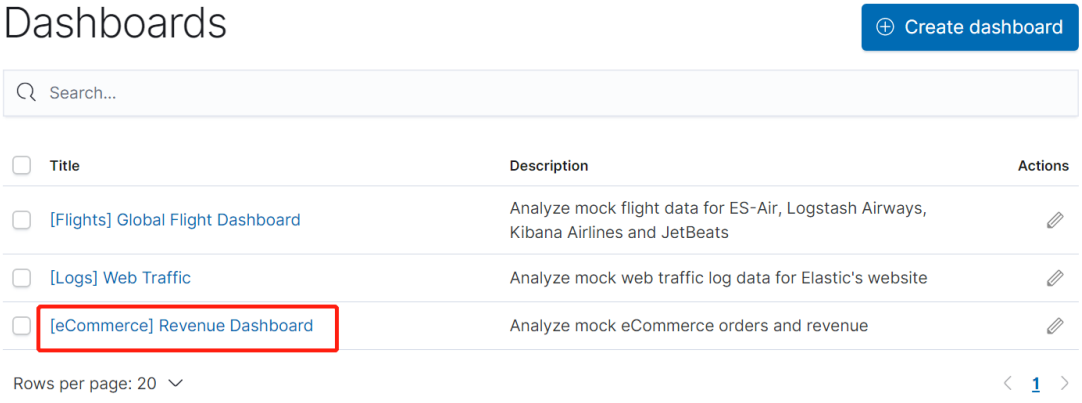
A figure as below can be generated.



Create a dashboard by selecting “Dashboard” from the Kibana Home page.



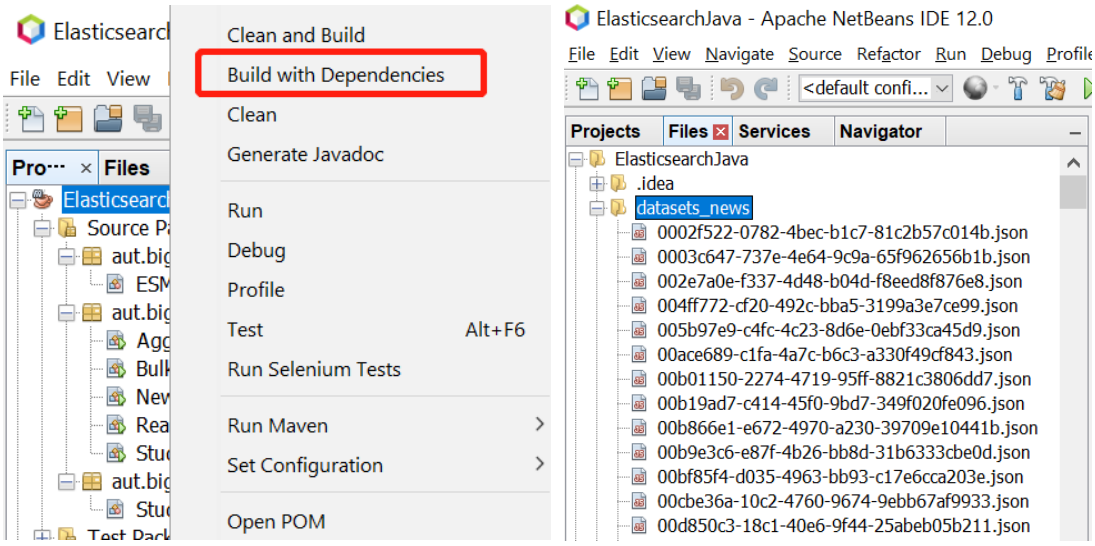
Select [eCommerce] Revenue Dashboard. You will see the pre-defined dashboard as below. There are more things for you to explore, please search and find by yourself if you are interested in.



[eCommerce] Revenue Dashboard:

Task 5 Elasticsearch – Java API (Optional)

Download ElasticsearchJava Project from the Canvas and open it using NetBeans 12. It is a Maven project. Right-click the project and select build with dependencies. All the associated java libraries will be downloaded.



Expand aut.bigdata.elasticsearck package and open ESManager.java file. The Elasticsearch connection configuration has been encoded here. It also provides basic functions for establishing connectivity and closing the connection. As Singleton Design pattern has been applied, only one Elasticsearch connection is allowed. You need to use the static public function *getESClient()* to obtain an instance.

Expand aut.bigdata.model package and open Student.java file. This is a java bean. Have a look at this file.

Expand aut.bigdata.examples, open StudentES.java. Walkthrough all the functions, including insert, update, delete, retrieval. Go to the main function, run it by right-clicking and selecting Run File.

In the same package, open BulkIngestion.java. Learn how to conduct bulk ingestion.

Check Aggregation.java and learn how to create a search request and build term aggregation.

ReadJSON.java utilises Jackson converting JSON files to map. This is a basic function for NewsIngestion.java, where the program converts all the news JSON files (crawled using News API) to map and ingest into Elasticsearch. The dataset has been incorporated in the project and you can find it from ElasticsearchJava\datasets_news

Question: Is there any other way of making the ingestion (program) more efficient?

Reference and Resources

- Elasticsearch Reference [7.x] » REST APIs » Document APIs, https://www.elastic.co/guide/en/elasticsearch/reference/7.x/docs.html
- Query DSL (Domain Specific Language), https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl.html
- Elasticsearch Queries: A Thorough Guide, Jurgens du Toit, Aug 16, 2020, https://logz.io/blog/elasticsearch-queries/
- Build your own dashboard, https://www.elastic.co/guide/en/kibana/7.9/tutorial-build-dashboard.html
- A Basic Guide To Elasticsearch Aggregations, Daniel Berman, Aug 29, 2019, https://logz.io/blog/elasticsearch-aggregations/
- Elasticsearch Java API, https://www.elastic.co/guide/en/elasticsearch/client/java-api/current/index.html
- Explore Kibana using sample data, https://www.elastic.co/guide/en/kibana/current/tutorial-sample-data.html
- Text similarity search with vector fields, Julie Tibshirani, Aug 28, 2019, https://www.elastic.co/blog/text-similarity-search-with-vectors-in-elasticsearch