



# Elastic Search

## A Search Engine

---

David THIBAU – 2022

[david.thibau@gmail.com](mailto:david.thibau@gmail.com)



# Agenda

---

- Introduction
  - ELK proposition and its use cases
  - Components of the stack
  - Distributions and installation
  - ELS API conventions, The DevConsole of Kibana
  - Core concepts of clustering
- Indexation and Documents
  - What is a document?
  - The API Document
  - Routing and distributed search
  - Search Lite
- Data ingestion
  - Beats concepts, provided beats by Elastic,
  - Logstash Concepts: pipelines, input, filters and output plugins
  - DSL syntax for configuring pipelines
  - Elastic search pipelines
- Index configuration
  - Supported Data types
  - How to control mapping?
  - Analyzers
  - Tuning Analyzer and filters
  - Reindexing
- Query syntax
  - DSL syntax
  - Main operators
  - Sorting and relevance score
  - Advanced search
  - Highlighting
  - Aggregations or faceting
  - Geo queries
- Elastic Search Clients
  - Clients libraries
  - Kibana's vizualisation and dashboards



# Introduction

---

## **ELK proposition and use cases**

Distributions and installation

ELS API conventions, The DevConsole  
of Kibana

Core concepts of clustering



# Introduction

---

The elastic company provides the ***ElasticStack suite*** in 2 editions (OpenSource and Enterprise)

The suite addresses 2 aspects, which can be applied in a BigData environment :

- **Near Real-time search**

Permanently indexed data is available for search, the latency due to indexing is negligible (Near Real Time)

- **Near Real-time data analysis.**

Data is aggregated in real time to provide visualizations and dashboards to extract insights

The core component of this stack is ***ElasticSearch***



# *ElasticSearch*

---

ElasticSearch is a server offering a REST API :

- To store and index data  
(Office documents, tweets, log files, performance metrics, ...)
- To perform search queries (structured, full-text, natural language, geolocation)
- Aggregate data in order to calculate metrics



# Features

---

- ✓ Massively **distributed and scalable** architecture in volume and load  
=> Big Data, remarkable performance
- ✓ **High availability** through replication
- ✓ **Muti-tenancy** or multi-index. The document base contains several indexes whose life cycles are completely independent
- ✓ Based on the **Lucene** reference library  
=> Full-text search, consideration of languages, natural language, etc.
- ✓ **Flexibility** : Structured storage of documents but without prior schema, possibility of adding fields to an existing schema
- ✓ Very complete **RESTFul API**
- ✓ **Open Source**



# Apache Lucene

---

- Apache project Started in 2000, used in many products
- The "low layer" of ELS: a Java library for writing and searching **index files**:
  - An index contains **documents**
  - A document contains **fields**
  - A field consists of **terms**

ELS brings scalability, a REST API, simplicity and aggregation features





# ELS vs SolR

The Apache Foundation offers **SolR** which is very close to ELS in terms of functionality

|   | SOLR                                    | ELS  |
|---|---|--|
| <b>Installation &amp; Configuration</b> | Not simple, Very detailed documentation | Simple and intuitive                       |
| <b>Indexation/ Search</b>               | Natural language oriented               | Text and other data types for aggregations |
| <b>Scalability</b>                      | Clustering via ZooKeeper and SolRCloud  | Natively cluster                           |
| <b>Community</b>                        | Important but stagnant                  | Has exploded                               |
| <b>Documentation</b>                    | Very complete but very technical        | Very complete, many examples               |





# Elastic Stack

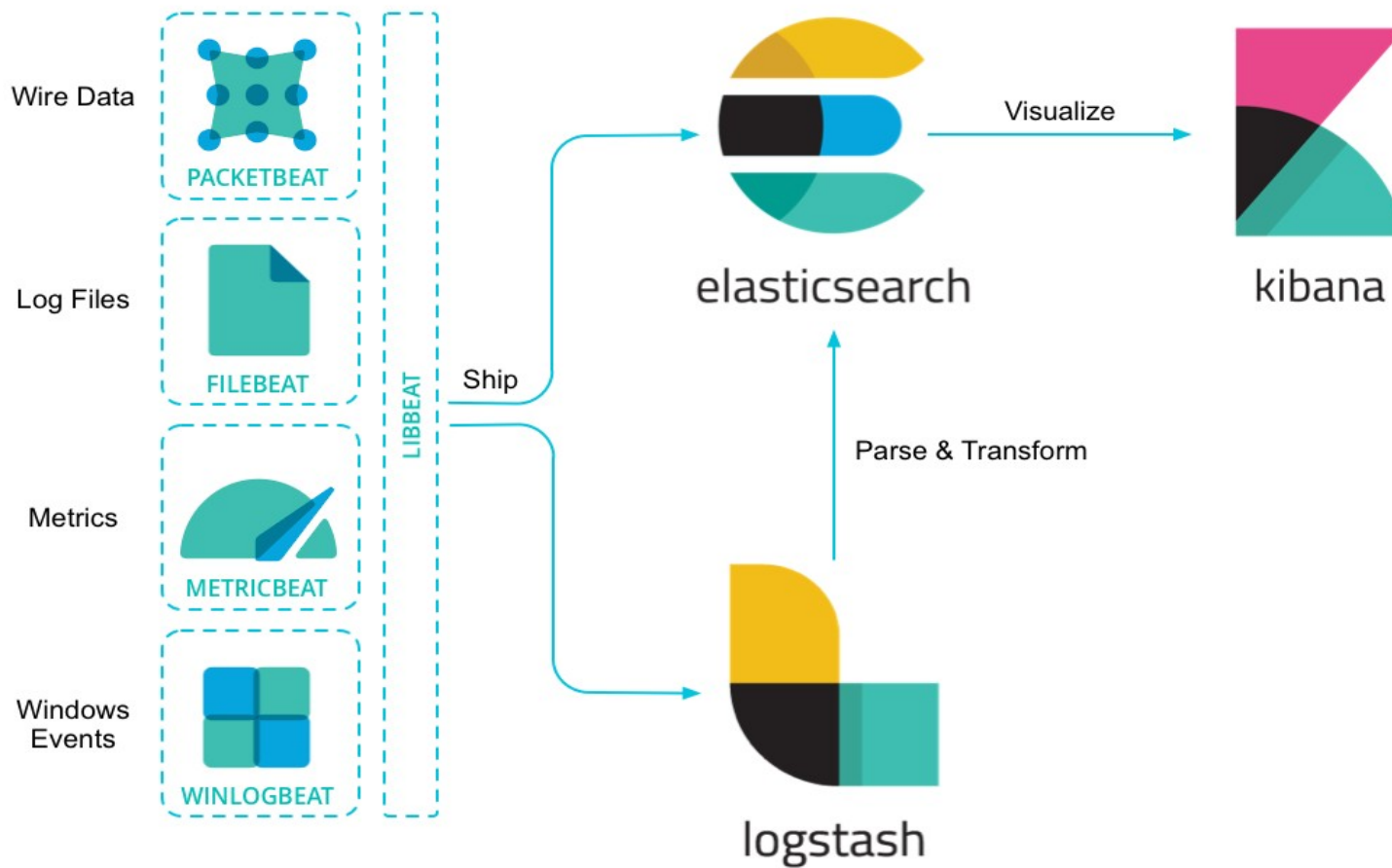
---

The suite Elastic Stack is dedicated to Real Time Analysis in the context of BigData.

It consist of :

- ***Logstash / Beat*** : Ingestion of data.  
Extract and transform
- ***Elastic Search*** : Storage, Indexing and Search
- ***Kibana*** : Data visualization

# Architecture





# Related Offers

---

The company Elastic also offers:

- ***X-Pack*** : commercial and enterprise features
  - Sécurité, Monitoring (before release 8.x)
  - Alerts, Machine Learning
  - Application Performance Management
  - Reporting and scheduling
- ***Elastic Cloud*** : Delegate operations of your cluster to the elastic company or Deploy elastic search in your private cloud



# Solutions out-of-the-box

---

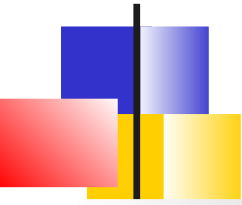
***Enterprise Search*** : Define sources of content (web, other..), index content and provides search capabilities

***Elastic Observability*** : Monitor CPU and memory consumption, aggregate logs, inspect network traffic, Application Performance Management .

Machine Learning to detect anomalies and anticipate future

***Elastic Security*** : Detect and respond to threats

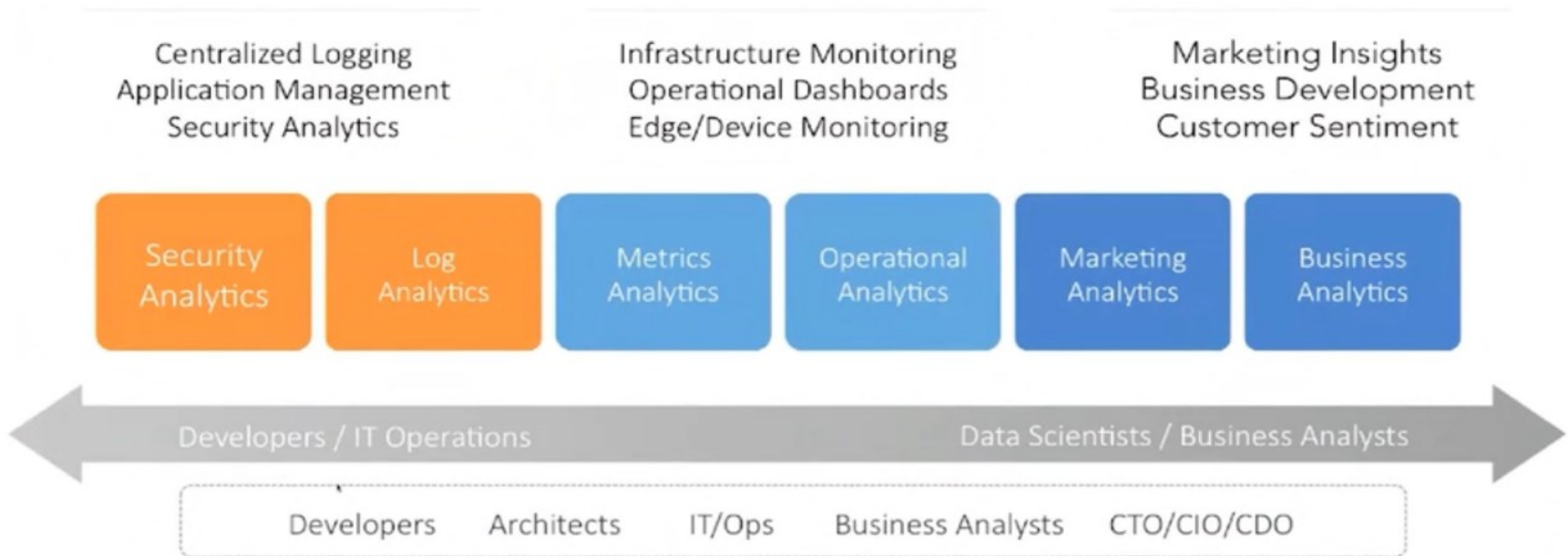
# Other Use Cases



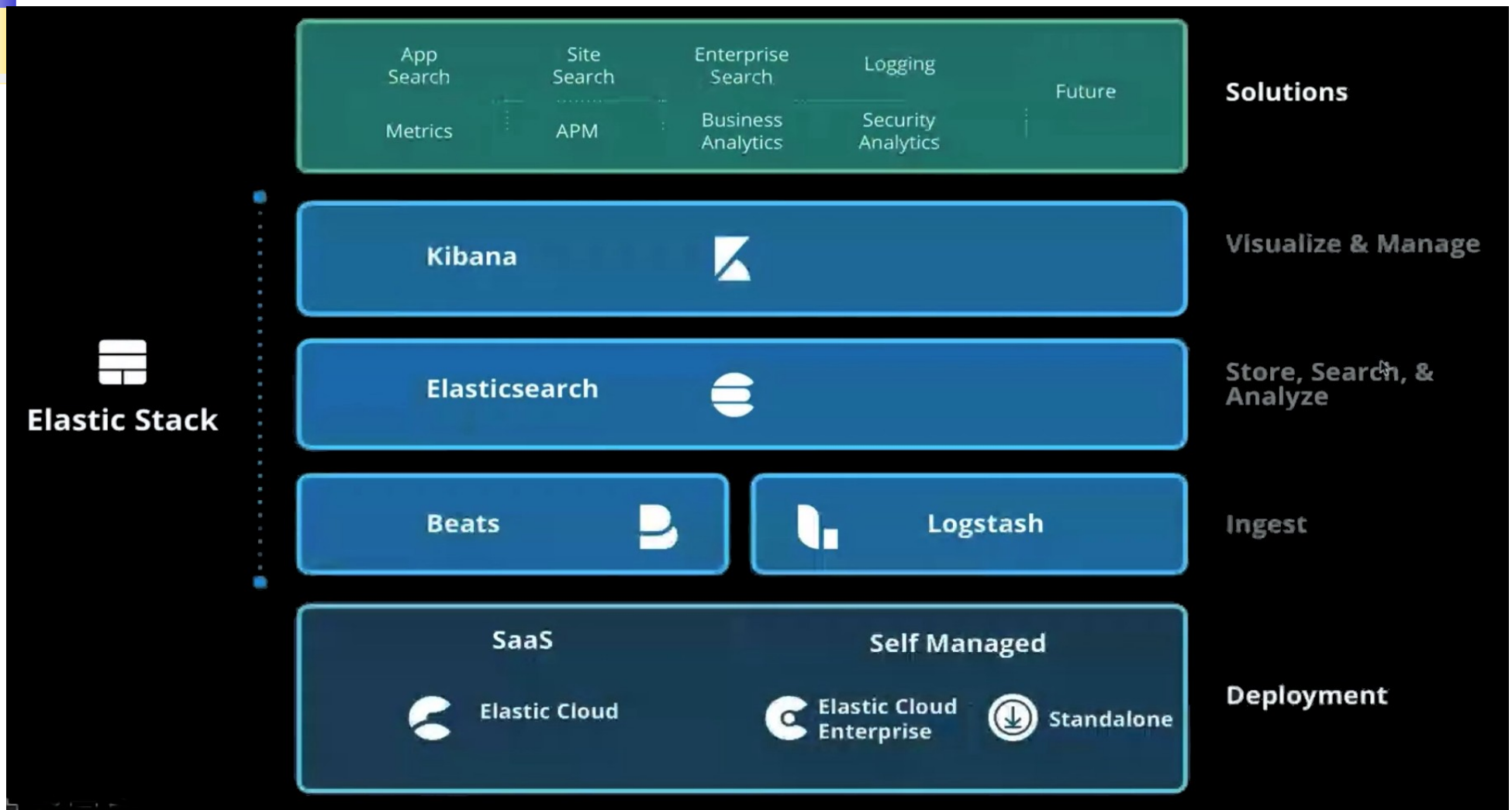
The stack can have very different usages

- Business Analytics : Real-time analysis of KPI
- Marketing : User behavior, frequentation analysis, marketing segmentation
- Business risk management and fraud detection
- IoT (domotic or health sensors ...)
- Big Data in general

# Elastic Use cases



# In summary





# Introduction

---

ELK proposition and use cases

**Distributions and installation**

ELS API conventions, The DevConsole  
of Kibana

Core concepts of clustering





# Releases

---

Releases are available for download in different formats :

- Archive ZIP, TAR,
- Packages DEB or RPM
- Docker Images

## Versions

- 8.0.0 : February 2022
- 7.x : Avril 2019 - Continuing
- 6.x : November 2017 to December 2019
- 5.0.0 : October 2016
- 2.4.1 : September 2016
- 2.4.0 : August 2016



# Installation ELS 8.x

---

1. Unzip the archive

2. Start ELS

`bin/elasticsearch`

3. Note the elastic user password  
and Access to ELS

`curl https://localhost:9200/`

# Configuration



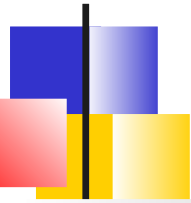
Several configuration files

- ***elasticsearch.yml*** : The property of the ELS node
- ***jvm.options*** : JVM Options
- ***log4j2.properties*** : Logging configuration

The default configuration allows to start quickly

In production, it is necessary to modify some parameters.  
For instance :

- The name of the cluster ***cluster.name***
- The name of the node : ***node.name***
- The public IP : ***network.host***
- Paths, in particular where to store data : ***paths***



# *Yet Another Markup Language*

---

YAML file is used for properties configuration

The syntax uses **:** and **indentation**

```
path:
```

```
  data: /var/lib/elasticsearch
```

```
  logs: /var/log/elasticsearch
```

Is equivalent to :

```
path.data: /var/lib/elasticsearch
```

```
path.logs: /var/log/elasticsearch
```



# *elasticsearch.yml*

---

ELS add some extra features to the  
YAML format :

- Environment variables d'environnement can be used with `${ENV_VAR}`
- Some properties may be prompted at startup:  
`${prompt.text}` , `${prompt.secret}`
- Values can be overridden at startup with the  
-E option  
`./elasticsearch -E node.name=David`



# *Bootstrap checks*

---

On startup ELS performs checks on the environment. If these checks fail :

- In development mode (IP listenning to localhost), warnings appear in the log
- In production mode (IP with a public adress), ELS refuse to start

These checks include, among other things:

- Heap sizing to avoid resizing and swapping
- Limit on the number of file descriptors very high (65,536)
- Allow 2048 threads
- Virtual memory size and areas (for native Lucene code)
- Filter on system calls
- Checking on behavior during JVM error or OutOfMemory



# Introduction

---

ELK proposition and use cases  
Distributions and installation  
**ELS API conventions, The  
DevConsole of Kibana**  
Core concepts of clustering



# ELS API

---

Typical Restful API with JSON

API is divided into category

- **document** (CRUD on documents)
- **index** (CRUD on Indexes)
- search (*\_search*)
- **cluster** monitoring (*\_cluster*)
- API **cat** : Tabbed response format for cluster management (*\_cat*)
- API **X-Pack** : Configuring X-Pack Features
- ...





# Common conventions

---

The different APIs follow a set of common conventions

- Ability to work on multiple indexes
- Date calculation functions to specify index names.
- Common options/parameters



# Multiple indexes

---

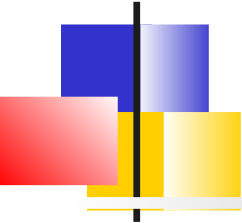
Most APIs that reference an index can be done across multiple indexes:

test1, test2, test3

\*test

+test\*, -test3

\_all



# Index names with Date Math

---

Index resolution with Date Math allows you to restrict the indexes used based on a date.

Indexes must be named with dates

Syntax is :

**<static\_name{date\_math\_expr{date\_format|time\_zone}}>**

- date\_math\_expr : xpression that computes a date
- date\_format : Format of rendering
- time\_zone : Time zone

Examples :

GET /<logstash-{now/d}>/\_search => **logstash-2017.03.05**

GET /<logstash-{now/d-1d}>/\_search => **logstash-2017.03.04**

GET /<logstash-{now/M-1M}>/\_search => **logstash-2017.02**



# Common parameters (1)

---

Parameter use **underscore casing**

**?pretty=true** : JSON well formatted

**?format=yaml** : yaml format

**?human=false** : If the response must be processed by a tool

**?filter\_path** : Response filter, allowing to specify response data

Ex : GET `/_cluster/state?pretty&filter_path=nodes`



## Common parameters (2)

---

**?flat\_settings=true** : Settings are returned using the notation "." rather than nested notation

**?error\_trace=true** : Inclut la stack trace dans la réponse

Time units : **d, h, m, s, ms, micros, nanos**

Size units : **b, kb, mb, gb, tb, pb**

Number without units : **k, m, g, t, p**

Distance unit : **km, m, cm, mi, yd, ft**

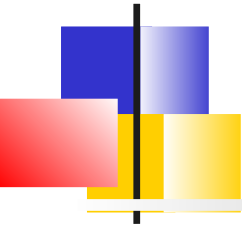


# Clients Alternatives

---

- Classical REST client:  
*curl*, Postman, SOAPUI, ...
- Libraries provided by Elastic :  
*Java, Javascript, Python, Groovy, Php, .NET, Perl*  
These libraries allow load-balancing on cluster nodes
- Finally, the most comfortable client is  
***Kibana's Dev Console***

# *Kibana's DevConsole*



The DevConsole contains 2 tabs:

- The editor tab for composing queries
- The response tab

The console understands a syntax close to Curl

```
GET /_search
```

```
{  
  "query": {  
    "match_all": {}  
  }  
}
```



# Features

---

- The console allows translation of CURL commands into its syntax
- It allows auto-completion, auto-indentation
- Passing on a single query line (useful for BULK queries)
- Allows you to run multiple queries
- Can change Elasticsearch server
- Keyboard shortcuts
- Search history
- It can be disabled (`console.enabled: false`)





# Kibana's Installation

---

Kibana version must strictly match  
ElasticSearch version

The distribution also includes the correct  
version of Node.js

It is available in different formats:

- Compressed Archive
- debian or rpm package
- Docker image

# Installation from an archive



---

```
wget  
https://artifacts.elastic.co/downloads/kibana/kibana-  
<version>-linux-x86_64.tar.gz  
sha1sum kibana-<version>-linux-x86_64.tar.gz  
tar -xzf kibana-<version>-linux-x86_64.tar.gz  
cd kibana/  
./bin/kibana
```



# Aborescence

---

***bin*** : Executable scripts including startup script and plugin install script

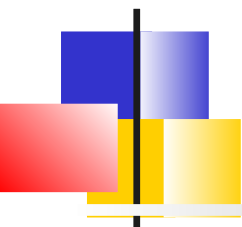
***config*** : Configuration files including *kibana.yml*

***data*** : Default data location, used by Kibana and plugins

***optimize*** : Translated code.

***plugins*** : Location of plugins. Each plugin corresponds to a directory

# Configuration



Main configuration properties defined in ***conf/kibana.yml*** :

- ***server.host, server.port*** : default localhost:5601
- ***elasticsearch.url*** : default <http://localhost:9200>
- ***kibana.index*** : Elasticsearch index to store kibana data
- ***kibana.defaultAppId*** : The home page of the UI
- ***logging.dest*** : Log file. Default *stdout*
- ***logging.silent, logging.quiet, logging.verbose*** : Logging level



# Introduction

---

ELK proposition and use cases  
Distributions and installation  
ELS API conventions, The DevConsole  
of Kibana  
**Core concepts of clustering**



# Cluster

---

A **cluster** is a set of servers (nodes) that contains all the data and offers search capabilities on the different nodes

- It has a unique name in the local network (default : "*elasticsearch*").

=> A cluster can consist of only one node

=> A node cannot belong to 2 separate clusters



# Node

---

A node is a single server (Java process) that is part of a cluster.

A node generally stores data, and participates in the indexing and search functionalities of the cluster.

A node is also identified by a unique name (generated automatically if not specified)

The number of nodes in a cluster is not limited



# Master node

---

In a cluster, a node is elected as the **master node**, it is in charge of managing the configuration of the cluster and the creation of indexes.

For all document operations (indexing, searching), each data node is interchangeable and a client can address any of the nodes

In large architectures, nodes can be specialized for data ingestion, Machine Learning processes, and in this case, they do not participate in search functionalities.





# Index

---

An index is a collection of documents that have similar characteristics

- For example an index for customer data, another for the product catalog and yet another for orders

An index is identified by a name (in lower case)

- The name is used for indexing and search operations

In a cluster, you can define as many indexes as you want



# Document

---

A **document** is the basic unit of information that can be indexed.

- document has a set of fields (key/value) expressed with JSON format

Inside an index, you can store as many documents as you want



# Shard

---

An index can store a very large amount of documents that may exceed the limits of a single node.

- To overcome this problem, ELS allows you to subdivide an index into several parts called ***shards***

When creating the index, it is possible to define the number of shards

Each shard is an independent index that can be hosted on one of the cluster nodes



# Apports du sharding

---

Sharding allows :

- To scale the volume of content
- To **distribute** and parallelize operations => improve performance

The internal mechanism of distribution during indexing and result aggregation during a search is completely managed by ELS and therefore transparent to the user



# Replica

---

To tolerate failures, it is recommended to use fail-over mechanisms in the event that a node fails (Hardware failures or upgrade of a system)

ELS allows you to set up copies of shards: **replicas** in order to

- **High-availability** : A replica is hosted on a different node than the primary shard
- To **scale** . Requests can be load balanced on every replica .



# Summary

---

In summary each index can be divided on several shards.

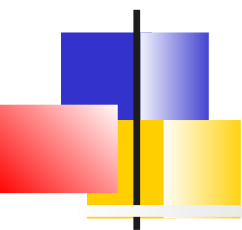
An index has also a replication factor

Once replicated, each index has primary shards and replication shards

Number of shards and replicas can be specified at index creation time

After its creation, the number of replicas can be changed dynamically but not the number of shards

Default for ELS 7.x+ : 1 shard and 1 replica



# Indexation and Documents

---

## **What is a document?**

The API Document

Routing and distributed search

Search Lite



# Introduction

---

*ElasticSearch* is a distributed **document** database.

It is able to store and retrieve data structures (in JSON format)

- Documents are made up of **fields**.
- Each field has a **data type**
- Some text fields are **analyzed and indexed** other not





# Sample

---

```
{
  "name": "John Smith",      // String (may be analyzed or not)
  "age": 42,                 // Nombre
  "confirmed": true,         // Booléen
  "join_date": "2014-06-01", // Date
  "home": {                 // Imbrication
    "lat": 51.5,
    "lon": 0.1
  },
  "accounts": [             // tableau de données
    {
      "type": "facebook",
      "id": "johnsmith"
    }, {
      "type": "twitter",
      "id": "johnsmith"
    }
  ]
}
```



# Metadata

---

Metadata is also associated with each document.

Main metadata are :

- ***\_index*** : The location where the document is stored
- ***\_id*** : The unique identifier
- ***\_version*** : The version number of the document
- ...



# Indexation and Documents

---

What is a document?

**The Document API**

Routing and distributed search

Search Lite



# Introduction

---

The document API enables CRUD operations on documents :

- Creation: *POST*
- Fetch by ID : *GET*
- Update : *PUT*
- Deletion : *DELETE*



# Indexation and *id*

---

During indexing (insertion in the database), it is possible :

- To provide an ID
- or to let ELS generate one



# Example with Id

---

**POST** `/ {index} /_doc/ {id}`

```
{  
  "field": "value",  
  ...  
}  
...  
{  
  "_index": {index},  
  "_id": {id},  
  "_version": 1,  
  "created": true  
}
```



# Example without Id

---

**POST** `/_{index}/_doc/`

```
{
  "field": "value",
  ...
}
...
{
  "_index": {index},
  "_type": {type},
  "_id": "wM00SFhDQXGZAWDf0-drSA",
  "_version": 1,
  "created": true
}
```



# Retrieving a document

---

To retrieve a document just provide its metadata (index and id) :

**GET /{index}/\_doc/{id}?pretty**

The response contains the document and its metadata.

Example :

```
{  "_index" : "website",
    "_type" : "blog",
    "_id" : "123",
    "_version" : 1,
    "found" : true,
    "_source" : {
        "title": "My first blog entry",
        "text": "Just trying this out...",
        "date": "2014/01/01"    } }
```





# Parts of a document

---

It is possible to retrieve only parts of the document.

Examples :

**# Metadata + title and text fields**

GET /website/\_doc/123?\_source=title,text

**# Just the source part without metadata**

GET /website/\_doc/123/\_source

**# Check if a document exist (Status 200 or 404)**

HEAD /website/\_doc/123



# Updates

---

Documents stored by ELS are immutable.

=> Updating a document consists of indexing a new version and deleting the old one.

The deletion is asynchronous and is done in batch mode (deletion of all replicas)



# Updating a document

---

**PUT /website/\_doc/123**

```
{  
  "title": "My first blog entry",  
  "text": "I am starting to get the hang of this...",  
  "date": "2014/01/02"  
}  
...  
{  
  "_index" : "website",  
  "_type" : "blog",  
  "_id" : "123",  
  "_version" : 2,  
  "created": false  
}
```



# Creation ... if not exist

---

2 syntaxs are possible to create a document only if it does not already exist in the database:

```
PUT /website/_doc/123?op_type=create
```

Or

```
PUT /website/_doc/123/_create
```

If the document exists, ELS responds with a 409 return code



# Deleting a document

---

```
DELETE /website/_doc/123
```

```
...
```

```
{
```

```
  "found" : true,
```

```
  "_index" : "website",
```

```
  "_type" : "blog",
```

```
  "_id" : "123",
```

```
  "_version" : 3
```

```
}
```



# Partial update

---

Documents are immutable: they cannot be changed, only replaced

- Updating fields therefore consists of re-indexing the document and deleting the old version

The ***\_update*** API can use the ***doc*** parameter to merge provided fields with existing fields



# Example

---

POST /website/\_update/1

```
{  
  "doc" : {  
    "tags" : [ "testing" ],  
    "views": 0  
  }  
}  
...  
{  
  "_index" : "website",  
  "_id" : "1",  
  "_version" : 3  
}
```



# Using scripting

---

A script (Groovy or Painless) can also be used to change the content of the `_source` field using a context variable:

***ctx.\_source***

Examples :

```
POST /website/_update/1 {
  "script" : "ctx._source.views+=1"
}
POST /website/_update/1 {
  "script" : {
    "source" : "ctx._source.tags.add(params.new_tag)",
    "params" : {
      "new_tag" : "search"
    }
  }
}
```

N.B Scripts can also be loaded from the particular `.scripts` index or configuration file.

They can be used in other contexts than an update.





# Bulk API

---

The **bulk** API allows you to perform several update orders (creation, indexing, update, deletion) in a single request

- => It is the batch mode of ELS.

But : Each request is handled separately. The failure of one request does not affect the others.

The Bulk API therefore cannot be used to set up transactions.



# Body of the request

---

The format of the request consist of several JSON documents, one on each line :

```
{ action: { metadata }}\n
```

```
{ request body }\n
```

```
{ action: { metadata }}\n
```

```
{ request body } \n
```

...

- Each line (even the last one) must be ended by  
\n
- Lines cannot contain other \n  
=> Then JSON documents can not be pretty formatted



# Syntax

---

An « action » line specifies :

- The action :
  - ***create*** : Création d'un document non existant
  - ***index*** : Création ou remplacement d'un document
  - ***update*** : Mise à jour partielle d'un document
  - ***delete*** : Suppression d'un document.
- The metadata : *\_id*, *\_index*



# Example

---

```
POST /website/_bulk // website is the default index
{ "delete": { "_type": "_doc", "_id": "123" }}
{ "create": { "_index": "website2", "_type": "_doc", "_id":
"123" }}
{ "title": "My first blog post" }
{ "index": { "_type": "_doc" }}
{ "title": "My second blog post" }
{ "update": { "_type": "_doc", "_id": "123"} }
{ "doc" : {"title" : "My updated blog post"} }
```



# Response

---

```
{
  "took": 4,
  "errors": false,
  "items": [
    { "delete": { "_index": "website",
      "_type": "_doc", "_id": "123",
      "_version": 2,
      "status": 200,
      "found": true
    } },
    { "create": { "_index": "website",
      "_type": "_doc", "_id": "123",
      "_version": 3,
      "status": 201
    } },
    { "create": { "_index": "website",
      "_type": "_doc", "_id": "EiwfApScQiiy7TIKFxRCTw",
      "_version": 1,
      "status": 201
    } },
    { "update": { "_index": "website",
      "_type": "_doc", "_id": "123",
      "_version": 4,
      "status": 200
    } }
  ]
}
```



# Indexation and Documents

---

What is a document?

The Document API

**Routing and distributed search**

Search Lite



# Primary shard resolution

---

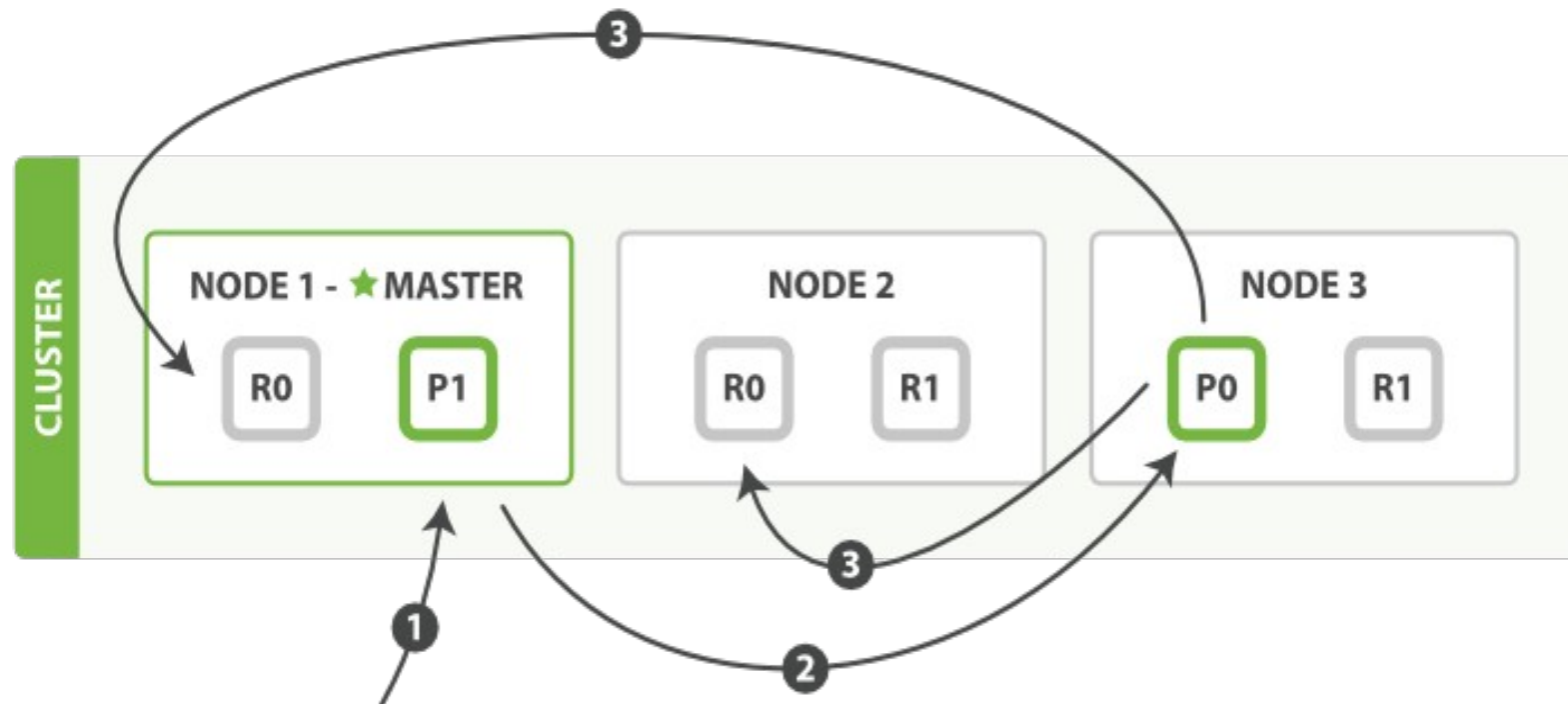
During indexing, the document is first stored in the primary shard.

Resolution of the shard is performed with the following formula:

```
shard = hash(routing) % number_of_primary_shards
```

The value of the ***routing*** parameter is an arbitrary string which by default matches the document ***id*** but which may be explicitly specified

# Sequence of updates during indexation





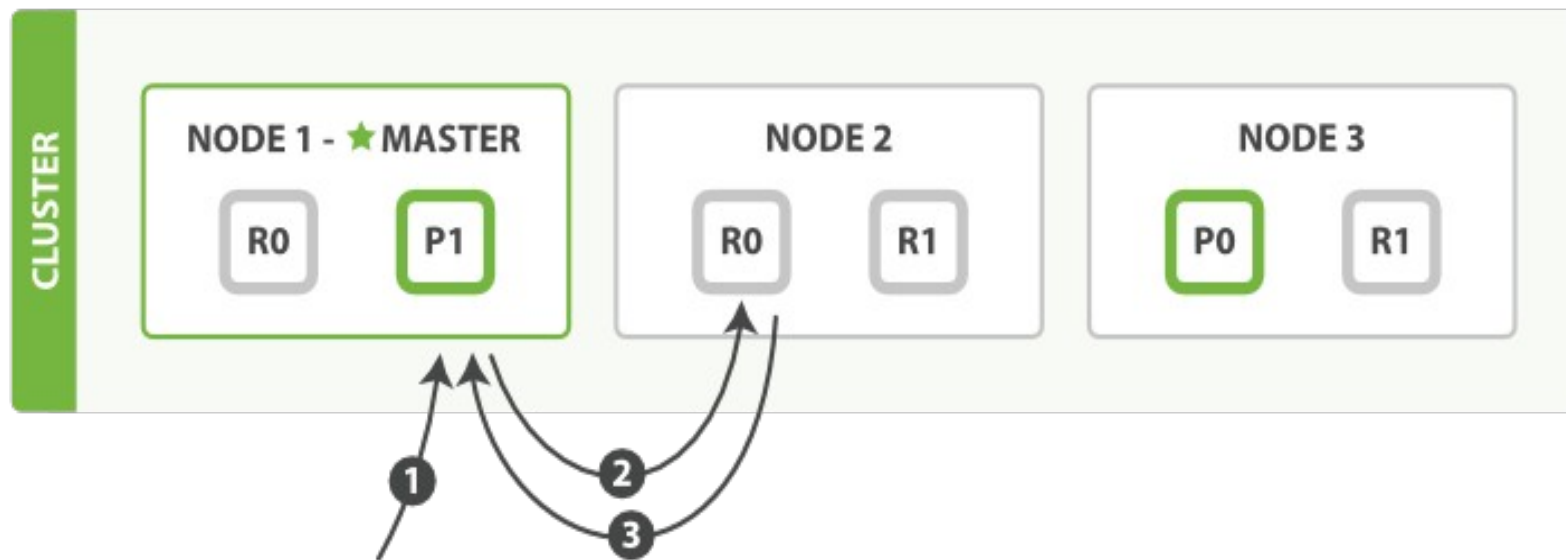


# Sequence of updates during indexation (2)

---

1. The client sends an indexing or deletion request to node 1
2. The node uses the document id to infer that the document belongs to shard 0 .  
It forwards the request to node 3 , where the primary copy of shard 0 resides.
3. Node 3 runs the query on the primary shard.  
If successful, it forwards the request in parallel to the replicas residing on node 1 and node 2 .  
Once the replica update orders succeed, node 3 responds to node 1 which responds to the client

# Sequence to retrieve a document





# Sequence to retrieve a document (2)

---

1. The client sends a request to node 1.
2. The node uses the document id to determine that the document belongs to shard 0. Copies of shard 0 exist on all 3 nodes.  
For this time, it forwards the request to node 2 .
3. Node 2 returns the document to node 1 which returns it to the client.

For the next read requests, the node will choose a different shard to distribute the load (Round-robin algorithm)



# Distributed search

---

Research requires a complex execution model, with documents scattered across shards

ElasticSearch must consult a copy of each shard of the index or indexes requested

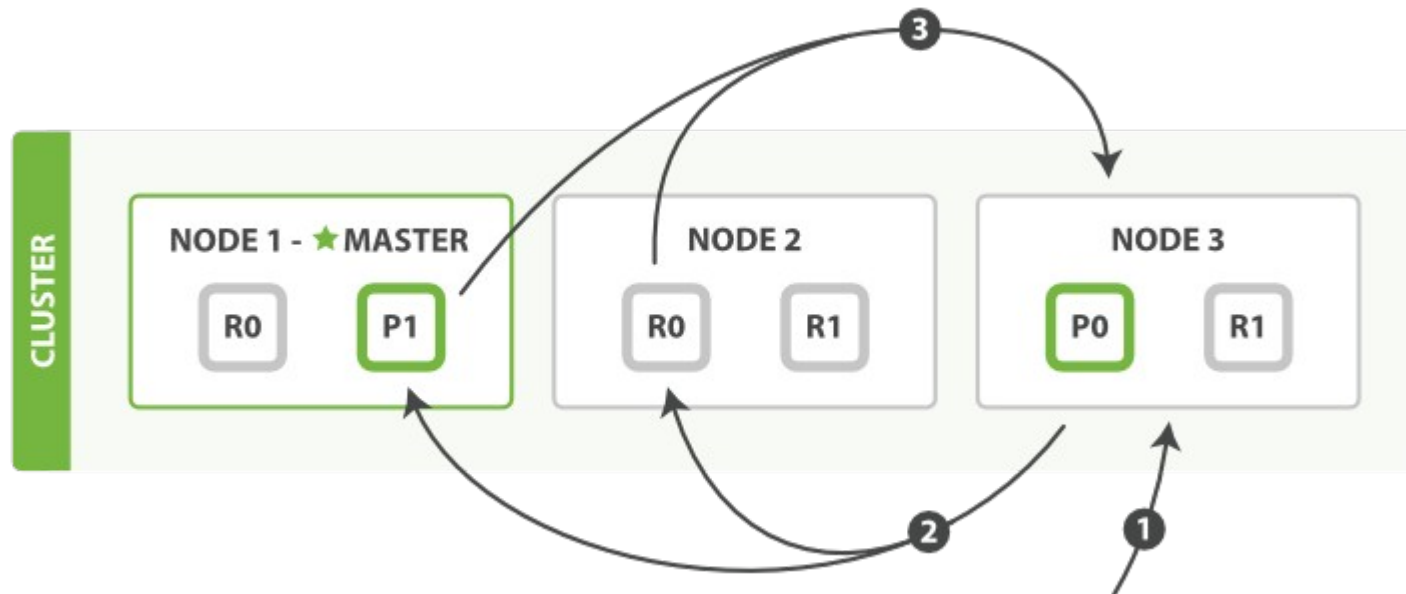
When found; results from different shards must be combined into a single list so that the API can return a page of results

The search is therefore executed in 2 phases:

- ***query***
- ***fetch.***

# Query phase

During the 1st phase, the request is broadcast to a copy (primary or replica) of all shards. Each shard performs the search and builds a priority queue of matching documents





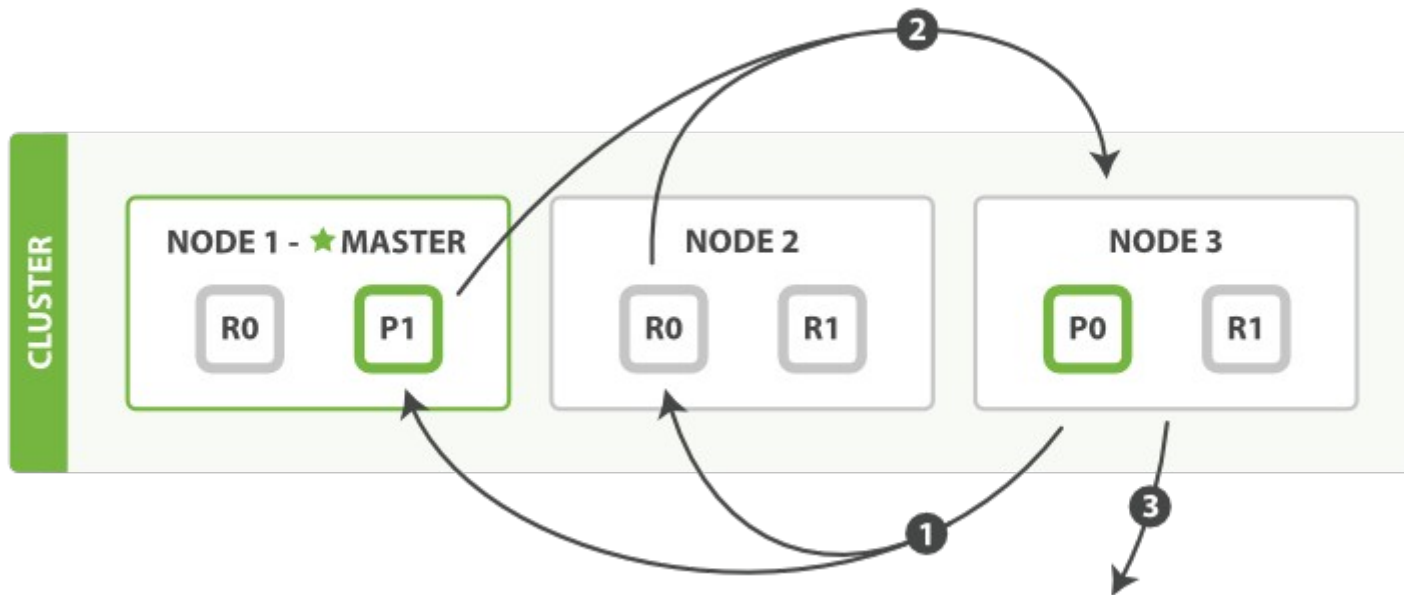
# Query phase

---

1. The client sends a query to node 3 which creates an empty priority queue of size = *from* + *size* .
2. Node 3 forwards the request to a copy of each shard in the index. Each shard executes the query locally and adds the result to a local priority queue of size = *from* + *size* .
3. Each shard returns the IDs and sort values of all documents in the queue to node 3 which merges these values into its priority queue

# Phase fetch

The fetch phase consists of retrieving the documents present in the priority queue.





# Fetch phase

---

1. The coordinator identifies which documents need to be retrieved and issues a multiple GET request to the shards.
2. Each shard loads the documents, enriches them if necessary (highlighting for example) and returns them to the coordinator node
3. When all documents have been retrieved, the coordinator returns the results to the client.





# Indexation and Documents

---

What is a document?

The Document API

Routing and distributed search

**Search Lite**



# Search APIs

---

## 2 Search APIs :

- A **lite** version where all parameters are provided with the query string.
- A **full** where search parameters are provided with the request body as a JSON structure. The query syntax is a DSL



# Search Lite

---

The lite version is however very powerful, it allows any user to execute heavy queries on all index fields.

This type of request can be a security hole allowing users to access confidential data or bring down the cluster.

=> In production, this type of API is generally prohibited in favor of DSL



# Empty query string

---

GET /\_search

No criteria : Search all documents of all indexes



# Response

---

```
{
  "hits" : {
    "total" : 14,
    "hits" : [ {
      "_index": "us",
      "_type": "tweet",
      "_id": "7",
      "_score": 1,
      "_source": {
        "date": "2014-09-17",
        "name": "John Smith",
        "tweet": "The Query DSL is really powerful and flexible",
        "user_id": 2
      }
    }, ... RESULTS REMOVED ... ],
    "max_score" : 1
  }, tooks : 4,
  "_shards" : {
    "failed" : 0,
    "successful" : 10,
    "total" : 10
  },
  "timed_out" : false
}
```



# Fields of the response

---

***hits*** : The number of documents that match the query, followed by an array containing the entire top 10 documents.

Each document has a `_score` element that indicates its relevance. By default, documents are sorted by relevance

***took*** : The number of milliseconds taken by the request

***shards*** : The total number of shards that participated in the request. Some may have failed

***timeout*** : Indicates if the request has timed out. The timeout parameter must have been provides like this :

GET `/_search?timeout=10ms`



# Limitation to specific index(es)

---

***/gb/\_search*** : All documents of  
index *gb*

***/gb,us/\_search*** : All documents of  
indexes *gb* and *us*

***/g\*,u\*/\_search*** : All documents of  
indexes starting with *g* or *u*



# Pagination

---

By default, only the first 10 documents are returned.

ELS accepts *from* and *size* parameters to control pagination:

- ***size*** : the number of documents which must be returned
- ***from*** : the indice of the firts returned document



# *q parameter*



The **q** parameter indicates the query string

The string is parsed and splitted in

- Fields : Fields of the document on which query is performed
- Terms : (Word or a phrase)
- And operators (AND/OR)

The syntax supports :

- Wildcards and regexp
- Grouping (parenthesis)
- Boolean operators (By default OR, + : AND, - : AND NOT)
- Intervals : Ex : [1 TO 5}
- Comparaison operators



# Examples search lite

---

GET /\_search?q=tweet:elasticsearch

All documents with the field *tweet* matches (~contains) "*elasticsearch*"

+name:john +tweet:mary

GET /\_search?q=%2Bname%3Ajohn+%2Btweet%3Amary

All documents whose the field *name* matches "john" and the field *tweet* matches "mary"

+name:john -tweet:mary

All documents whose the field *name* matches "john" and the field *tweet* does not matches "mary"



# Field *\_all*

---

When indexing a document, ELS concatenates all values of type string into a full-text field named ***\_all***

This is the default field used if the query does not specify a field

GET /\_search?q=mary

All documents whose one of its field *name* matches "mary"

If the *\_all* field is not useful, it is possible to disable it



# More complex example

---

The following query applies these criteria :

- The field *name* contains « mary » or « john »
- The date is greater than « 2014-09-10 »
- The field *\_all* contains either the word « aggregations » or « geo »

***+name:(mary john) +date:>2014-09-10 +  
(aggregations geo)***

See the full doc here :

<https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-query-string-query.html#query-string-syntax>



# Other parameters of the query string

---

Other available parameters are :

- **df** : The default field, used when no field is specified in the request
- **default\_operator** (AND/OR) : The default operator.  
Default OR
- **explain** : An explanation of the score or error for each hit
- **\_source** : *false* to avoid fetching the `_source` field in the response .  
Retrieved fields can also specified with `_source_include` & `_source_exclude`
- **sort** : Sort. For example *title:desc,\_score*
- **timeout** : Search timeout. When the timeout expires, the hits found are returned



# Data Ingestion

---

## **Beats**

Logstash pipelines

Logstash DSL

Main inputs, codecs, filters and  
outputs

Elastic search pipelines



# Introduction to Beats

---

Beats convey data

They are installed as agents on the different target (servers) and periodically send their data

- Directly to *ElasticSearch*
- Or, if the data need to be transformed, to *Logstash*



# Types of beats

---

ELK provides :

- ***Packetbeat*** : All the network packets seen by a specific host
- ***Filebeat*** : All the lines written in a log file .
- ***Metricbeat*** : Performance metrics on CPU, Memory, Disk, IO, etc
- ***Winlogbeat*** : Windows log events

They all produce data which can be directly ingested in ElasticSearch

Other beats provided by third party are also available





# Data Ingestion

---

Beats

**Logstash pipelines**

Logstash DSL

Main inputs, codecs, filters and  
outputs

Elastic search pipelines



# Introduction

---

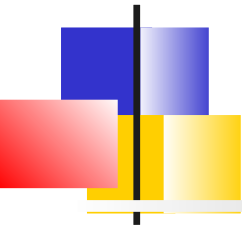
Logstash is a **real-time data collection** tool able to unify and normalize data from different sources

Originally focused on log files, it has evolved to handle very diverse events

Logstash is based on the notion of processing **pipelines**.

Extensible via plugins, it can read/write from/to very different sources and targets

# Sources of events



---

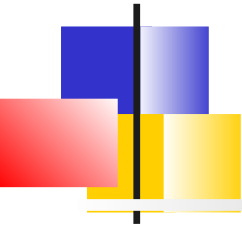
Traces and metrics : Server logs, syslog, Windows Event, JMX, ...

Web : HTTP request, Twitter, Hook for GitHub, JIRA, Polling d'endpoint HTTP

PersistenceStore : JDBC, NoSQL, \*MQ

Miscellaneous sensors : Mobile phones, Connected Home or Vehicles, Health sensors

# Data enrichment



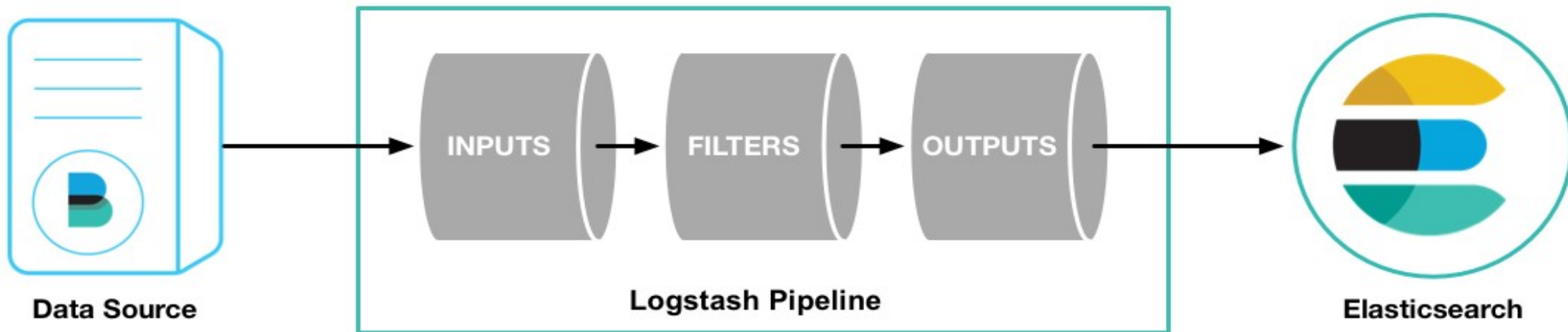
Logstash also allows you to enrich the input data.

- Geo-location from an IP address. (lookup to a service)
- Data encoding/decoding
- Date normalization
- Elastic Search queries for enrichment
- Anonymize sensitive information

# Pipeline Logstash

A logstash **pipeline** defines :

- An input plugin to read data from a source
- An output plugin to write data to a target (typically ElasticSearch)
- Optional filters to perform transformations



# Configuration

Pipelines are generally defined in files :

```
# The # character at the beginning of a line indicates a comment. Use
# comments to describe your configuration.
input {
}
# The filter part of this file is commented out to indicate that it is
# optional.
# filter {
#
# }
output {
}
```

The configuration can also be precised in the command line

```
bin/logstash -e 'input { stdin { } } output { stdout { } }'
```



# Sample

---

```
input {
  beats { port => "5043" }
}
filter {
  grok {
    match => { "message" => "%{COMBINEDAPACHELOG}" }
  }
  geoip {
    source => "clientip"
  }
}
output {
  elasticsearch {
    hosts => [ "localhost:9200" ]
  }
}
```



# Multiple inputs/outputs

---

*Logstash* is able to process multiple inputs and route them to multiple outputs

```
input {
  twitter {
    consumer_key => "enter_your_consumer_key_here"
    consumer_secret => "enter_your_secret_here"
    keywords => ["cloud"]
    oauth_token => "enter_your_access_token_here"
    oauth_token_secret => "enter_your_access_token_secret_here"
  }
  beats {
    port => "5043"
  }
}
output {
  elasticsearch {
    hosts => ["IP Address 1:port1", "IP Address 2:port2", "IP Address 3"]
  }
  file {
    path => "/path/to/target/file"
  }
}
```





# Event and fields

---

Events are processed by the pipeline.

An event can be :

- A line in a file
- A row in a database table
- A whole json or XML file

Generally, event is splitted into fields and each field may be transformed if needed.

A field's value can be a simple value or a structure (Map)



# Data Ingestion

---

Beats

Logstash pipelines

**Logstash DSL**

Main inputs, codecs, filters and  
outputs

Elastic search pipelines

# Introduction



---

The configuration file of a pipeline specifies plugins and their configuration

Each plugin listed has a configuration block specific to the plugin

Configuration values are typed

A specific syntax is used to reference a field of an event

Control structure may condition the execution of a plugin



# Supported Data Types

---

Following Data types are supported by plugins :

- Simple types : Boolean, numbre or String  
    `ssl_enable => true    name => "Hello world"`
- HashMap  
    `match => {     "field1" => "value1"  
                  "field2" => "value2" }`
- Size in bytes  
    `my_bytes => "10MiB"    # 10485760 bytes`
- Template string : Uri, password, path  
    `my_path => "/tmp/logstash"    my_uri => "http://foo:bar@example.net"  
    my_password => "password"`
- Predefined values (enumeration)  
    `codec => "json"`
- Comments  
    `# this is a comment`

# Referencing fields

To reference a top level field

- just specify the field. Eg: **agent**.
- Or use the notation []. Eg: **[agent]**

For nested fields, use the [ ] notation. Ex **[ua][os]**

```
{
  "agent": "Mozilla/5.0 (compatible; MSIE 9.0)",
  "ip": "192.168.24.44",
  "request": "/index.html"
  "response": {
    "status": 200,
    "bytes": 52353
  },
  "ua": {
    "os": "Windows 7"
  }
}
```

# Examples *sprintf*



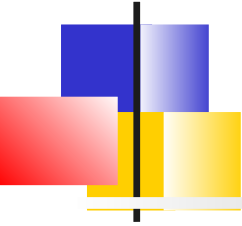
The **%{ }** notation, named ***sprintf***, is used to concatenate static values and field values

- A specific notation is used to retrieve today's date in a particular format.

```
output {  
  statsd {  
    increment => "apache.%{[response][status]}"  
  }  
}
```

```
output {  
  file {  
    path => "/var/log/%{type}.%{+yyyy.MM.dd.HH}"  
  }  
}
```

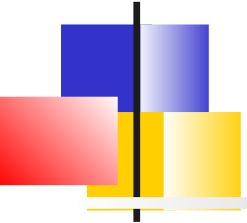
# Conditional configuration



Conditional configuration is performed with *if then else* instructions

```
if EXPRESSION {  
    ...  
} else if EXPRESSION {  
    ...  
} else {  
    ...  
}
```

# Examples



```
filter {  
  if [action] == "login" {  
    mutate { remove_field => "secret" }  
  } }  

```

```
output {  
  # Send production errors to pagerduty  
  if [loglevel] == "ERROR" and [deployment] == "production" {  
    pagerduty {  
      ...  
    } } }  

```

```
if [foo] in ["hello", "world", "foo"] {  
  mutate { add_tag => "field in list" }  
}
```

The expression `if [foo]` returns false if :

- `[foo]` does not exist in the event,
- `[foo]` exist but is false or null





# The field *@metadata*

---

It exist a special field : ***@metadata***

This field will not be written on the outputs, however it can be used during all Logstash processing

```
input { stdin { } }
```

```
filter {  
  mutate { add_field => { "show" => "This data will be in the output" } }  
  mutate { add_field => { "[@metadata][test]" => "Hello" } }  
  mutate { add_field => { "[@metadata][no_show]" => "This data will not be in the  
output" } }  
}
```

```
output {  
  if [@metadata][test] == "Hello" {  
    stdout { codec => rubydebug }  
  }  
}
```



# Environnement variables

---

An environment variable can be referenced by ***`${var}`***

- References to undefined variables cause errors
- Variables are case sensitive
- A default value can be provided with :  
*`${var: default value}`*.



# Data Ingestion

---

Beats

Logstash pipelines

Logstash DSL

**Main inputs, codecs, filters and  
outputs**

Elastic search pipelines



# Inputs

---

Some inputs :

- **beats** : Events produced by a beat
- **elasticsearch** : ElasticSearch Query response
- **exec** : Shell command output
- **file** : Reading lines of files
- **jdbc** : Records from database
- **http-poll, http, STOMP** : Events received via HTTP
- **Imap, Irc, rss** : Read mails, irc, rss, twitter
- **Kafka, redis, rabbitmq, jms** : Messages in message brokers
- **Tcp, udp, unix, syslog** : Low level socket events ...



# Examples

---

```
input {
  beats {
    port => 5044
  }
  file {
    path => "/var/log/*"
    exclude => "*.gz"
  }
  generator {
    lines => [
      "line 1",
      "line 2",
      "line 3"
    ]
    # Emit all the lines 3 times, useful for testing.
    count => 3
  }
}
```



# Some Codecs

---

Codecs are used to decode/encode data. They are used in input or output

- ***gzip\_lines*** : Unzip content encoded with gzip
- ***json*** : Read JSON well-formatted content and translate it in logstash fields.
- ***json\_lines*** : Read JSON formatted in one-line (Bulk API de ELS)
- ***multiline*** : Merge several lines into one event
- ***rubydebug*** : Use to debug, print logstash fields on stdout



# Multi-line events

---

During multi-row events, Logstash needs to know which rows should be part of the single event

Multi-line processing is usually done first in the pipeline and uses the ***multiline*** filter or codec

3 important configuration options:

- ***pattern*** : Regular expression. Matching lines are considered either as the continuation of previous lines or the start of a new event
- ***what*** : *previous* or *next*
- ***negate*** : Express the negation



# Example : XML File

---

```
<article id="5670">
  <author>Bill Smith</author>
  <content>
    The multiline filter allows to create xml file as a single event and we can use xml-filter or
    xpath to parse the xml to ingest data in elasticsearch
  </content>
</article>
```

The following filter indicate if the line match *<article>*, it must not be aggregated with the previous line

```
input {
  file {
    path => "/opt/inputFiles/*.xml"
    codec => multiline {
      pattern => "<article>"
      what => "previous"
      negate => "true"
    }
  }
}
```





# Some filters

---

***geoip*** : Add latitude/longitude information from an IP

***grok*** : Split a field into other fields via regular expressions

***mutate*** : Transform a field

***prune*** : Filter events from a black or white list

***range*** : Checks that the size or length of a field is in an interval

***translate*** : Replaces field values from a hashtable or YAML file

***truncate*** : Truncate fields greater than a certain length

***urldecode*** : Decode URL-encoded fields

***useragent*** : Parse user agent strings

***xml*** : Parses the XML format



# The mutate filter

---

The swiss-knife for transformations, configuration options are

- **convert** : Type conversion
- **copy** : Copy a field in another field
- **gsub** : Replacing character strings from regular expressions
- **lowercase** / **uppercase** : Switch to lowercase/uppercase
- **join, merge** : Work on arrays
- **rename** : Renaming a field
- **replace, update** : Replace a field's value
- **split** : Split a field in a array with a separator
- **strip** : Remove white spaces at the beginning and the end



# The *date* filter

---

The filter is used to parse dates from a field.  
The options are:

- ***match*** : An array whose first value is the parser field and the other values the possible formats
- ***locale***, ***timezone*** : If not present in the match format, we can specify
- ***target*** : The field storing the result, by default *@timestamp*



# Output plugins

---

## Some outputs

- **csv** : Write CSV files
- **elasticsearch** : Index into Elasticsearch
- **email** : Send the event via mail
- **file** : Writes into a file
- **stdout** : Writes in the standard output
- **pipe** : Send to standard input of another program
- **http** : Send to an HTTP or HTTPS endpoint
- **kafka, redis, rabbitMQ** : Messaging
- ...



# *Elasticsearch* Output

---

Main options of elasticsearch outputs are :

- **hosts** : Data nodes of the cluster
- **index** : The name of the index. Example : logstash-%  
{+YYYY.MM.dd}.
- **template** : A path to a template to create the index
- **template\_name** : The template's name present in Elasticsearch
- **document\_id** : The id of the Elasticsearch document (allow us  
updates)
- **pipeline** : The Elasticsearch pipeline to execute before  
indexing
- **routing** : To specify the shard to use



# Several pipelines

---

Logstash allows us to run multiple pipelines in the same process.

Different performance or durability configurations can be applied

The configuration is done by an additional file ***pipelines.yml*** placed in *path.settings*



# Example pipeline.yml

---

- pipeline.id: my-pipeline\_1  
path.config: "/etc/path/to/p1.config"  
pipeline.workers: 3
- pipeline.id: my-other-pipeline  
path.config: "/etc/different/path/p2.cfg"  
queue.type: persisted



# Data Ingestion

---

Beats

Logstash pipelines

Logstash DSL

Main inputs, codecs, filters and  
outputs

**Elastic search pipelines**





# Introduction

---

Without logstash, it is possible to perform common transformations on the data before indexing by using the **ingest pipelines of Elasticsearch**

Features are similar to logstash but less powerful

ELS pipelines can be used to remove fields, extract values from text, and enrich your data.

It consists of a series of configurable tasks called **processors**. Each processor runs sequentially, making specific changes to incoming documents



# Pipeline

---

A **pipeline** is defined via :

- A name
- A description
- And a list of *processors*

Processors are predefined

They will be executed in sequence



# Ingest API

---

The ***\_ingest*** API supports :

- ***PUT*** : Adding or replacing a pipeline
- ***GET*** : Retrieve a pipeline
- ***DELETE*** : Delete a pipeline
- ***SIMULATE*** : Simulate the execution of the pipeline



# Usage

---

**#Create a pipeline named *attachment***

```
PUT _ingest/pipeline/attachment
{
  "description" : "Extract attachment information",
  "processors" : [
    { "attachment" : { "field" : "data" } }
  ]
}
```

**# Using a pipeline during indexation of a document.**

```
PUT my_index/my_type/my_id?pipeline=attachment
{
  "data":
  "e1xydGYxXGFuc2kNCkxvcmVtIGlwc3VtIGRvbG9yIHNpdCBhbWV0DQpccGFy
  IH0="
}
```



# Result

---

GET my\_index/my\_type/my\_id

```
{
  "found": true,
  "_index": "my_index",
  "_type": "my_type",
  "_id": "my_id",
  "_version": 1,
  "_source": {
    "data":
    "e1xydGYxXGFuc2kNCkxvcmVtIGlwc3VtIGRvbG9yIHNpdCBhbWV0DQpccGFyIH0=",
    "attachment": {
      "content_type": "application/rtf",
      "language": "ro",
      "content": "Lorem ipsum dolor sit amet",
      "content_length": 28
    }
  }
}
```



# Introduction to plugins

---

The functionalities of ELS can be increased via the notion of plugin

Plugins contain JAR files, but also scripts and configuration files

They can be installed easily with

```
sudo bin/elasticsearch-plugin install [plugin_name]
```

- They must be installed on each node of the cluster
- After an installation, the node must be restarted



# Ingest plugins

---

A particular ingest plugins allow to index office documents (Word, PDF, XLS, ...) :

- ***Attachment Plugin*** : Extracts text data from attachments in different formats (PPT, XLS, PDF, etc.). It uses the Apache Tika library.

```
sudo bin/elasticsearch-plugin install ingest-attachment
```



# Index configuration

---

## **Supported Data types**

Text Analyzers

How to control mapping?

Tuning Analyzers and filters

Reindexing





# Introduction

---

Each field of a document has an associated datatype.

The metadata which specifies the different fields and their associated types is called the **mapping**

The mapping API is used to define, visualize, update the mapping of an index



# Supported simples data types

---

ELS supports the following simple data types :

- 2 kinds of string : *text* (analyzed) or *keyword* (not analyzed)
- Numbers : *byte* , *short* , *integer* , *long* , *float* , *double* , *token\_count*
- Booleans : *boolean*
- Dates : *date*
- Bytes : *binary*
- Range : *integer\_range* , *float\_range* , *long\_range* , *double\_range* , *date\_range*
- IP adress : *IPV4* ou *IPV6*
- Geolocation : *geo\_point* , *geo\_shape*
- A query (JSON structure) : *percolator*



# Complex data types

---

In addition to simple types, ELS supports

- **arrays** : There is no special mapping for arrays. Each field can contain 0, 1 or n values  

```
{ "tag": [ "search", "nosql" ] }
```

  - Array values must be of the same type
  - A null field is treated as an empty array
- **objects** : It is a data structure embedded in a field
  - Each sub-field is accessed with *object.property*
- **nested** : It is an array of data structures embedded in a control. Each array element is stored separately and has another *id*
  - It is generally a bad idea to use nested objects



# Mapping of embedded objects

ELS dynamically detects object fields and maps them as an object. Each embedded field is listed under **properties**

```
{ "gb": {  
  "properties": {  
    "tweet": { "type": "string" },  
    "user": {  
      "type": "object",  
      "properties": {  
        "id": { "type": "string" },  
        "age": { "type": "long"},  
        "name": {  
          "type": "object",  
          "properties": {  
            "first": { "type": "string" },  
            "last": { "type": "string" }  
          }  
        }  
      }  
    }  
  }  
}
```



# Exact value or full-text

---

*String* stored by ELS can be of 2 data types :

- **keyword** : The value is taken as is, filter type operators can be used when searching  
Foo!= foo
- **text** : The value is analyzed and split into terms or tokens. Full-text search operators can be used when searching. This concerns data in natural language



# Inverted index

---

In order to speed up searches on text fields, ELS uses a data structure called ***inverted index***

This consists of a unique word list where each word is associated with the documents in which it appears.

The method list is created after an analysis phase of the original text field.



# Example

---

|    | A           | B                   |
|----|-------------|---------------------|
| 1  | term        | docs                |
| 2  | pizza       | 3, 5                |
| 3  | solr        | 2                   |
| 4  | lucene      | 2, 3                |
| 5  | sourcesense | 2, 4                |
| 6  | paris       | 1, 10               |
| 7  | tomorrow    | 1, 2, 4, 10         |
| 8  | caffè       | 3, 5                |
| 9  | big         | 6                   |
| 10 | brown       | 6                   |
| 11 | fox         | 6                   |
| 12 | jump        | 6                   |
| 13 | the         | 1, 2, 4, 5, 6, 8, 9 |



# Index configuration

---

Supported Data types

**Text Analyzers**

How to control mapping?

Tuning Analyzers and filters

Reindexing





# Tokenization and Normalization

---

To build the inverted index, ELS needs to separate a text into words (**tokenization**) and then **normalize** the words so that searching for the term "sending" for example returns documents containing: "sending", "Sending", "sendings", "send", ...

Tokenization and normalization are called **analysis**.

The analysis applies to the documents during indexing **AND** during the search on the search terms.



# Analysis steps

---

Analyzers transform a text into a stream of “tokens”. It is a combination of:

- **Character filters** prepare text by performing character replacement (& becomes *and*) or deletion (removal of HTML tags)
- **Tokenizer** . It splits a text into a series of lexical units: tokens (only 1 per analyzer)
- **Filters** take a token stream as input and transform it into another token stream



# Predefined analyzers

---

ELS offers directly usable analyzers:

- **Standard Analyzer** : This is the default analyzer. The best choice when the text is in various languages. It consists of :
  - Split text into words
  - Remove punctuation
  - Lowercase all words
  - Supports stop words
- **Simple analyzer** : Separates text based on character different than a letter, changes to lowercase
- **Stop analyzer** : Idem as simple but supports stop words
- **WhiteSpace Analyzer**: Separates text based on spaces
- **Pattern analyzer** : Based on regular expression
- **Language analyzer**: These are language-specific parsers. They include "stop words" (remove most common words) and extract the root of a word. This is the best choice if the index is in one language
- **Fingerprint analyzer** : Use for duplicate detection



# Testing analyzers

---

Every analyzers can be tested via the API :

GET /\_analyze?analyzer=standard  
Text to analyze

Réponse :

```
{
  "tokens": [ {
    "token": "text",
    "start_offset": 0,
    "end_offset": 4,
    "type": "<ALPHANUM>",
    "position": 1
  }, {
    "token": "to",
    "start_offset": 5,
    "end_offset": 7,
    "type": "<ALPHANUM>",
    "position": 2
  }, {
    "token": "analyze",
    "start_offset": 8,
    "end_offset": 15,
    "type": "<ALPHANUM>",
    "position": 3
  } ] }
```



# Specify an analyzer

---

When ELS detects a new string field during indexation, it automatically stores it twice :

- As ***text*** field, and applies the standard parser.
- As ***keyword*** and store its exact value (length is limited to first 256 characters)

If this is not the desired behavior, you must explicitly specify the mapping for the document field before indexing



# Index configuration

---

Supported Data types  
Text Analyzers

**How to control mapping?**

Tuning Analyzers and filters  
Reindexing



# Mapping

---

ELS is able to dynamically generate the mapping during indexation

- It guesses the type of fields
- For string, it uses text and keyword datatypes

The mapping of an index can be retrieved with :

GET /gb/\_mapping



# Response

---

```
{ "gb": {  
  "mappings": {  
    "tweet": {  
      "properties": {  
        "date": {  
          "type": "date",  
          "format": "dateOptionalTime"  
        },  
        "name": {  
          "type": "text",  
          "fields": {  
            "keyword": {  
              "type": "keyword",  
            }  
          }  
        },  
        "tweet": {  
          "type": "text"  
        },  
        "user_id": {  
          "type": "long"  
        }  
      }  
    }  
  }  
}
```





# Define Mapping before indexation

---

Define mapping before indexing allows you to :

- Distinguish strings between text or keyword
- Use specific analyzers
- Define multiple analyzers for the same field
- Optimize a field for partial matching (see further)
- Specify custom date formats
- Specify data types not detectable by ELS,  
example *geo\_point*
- ...



# Define mapping

---

You can specify the mapping:

- When creating an index
- When adding a new field to an existing index

=> You cannot modify a field that is already indexed



# Example : index creation

---

PUT /gb

```
{
  "mappings": {
    "tweet" : {
      "properties" : {
        "tweet" : {
          "type" : "text",
          "analyzer": "english"
        },
        "date" : {
          "type" : "date"
        },
        "name" : {
          "type" : "keyword"
        },
        "user_id" : {
          "type" : "long"
        }
      }
    }
  }
}
```



# Example : Adding a new field

---

```
PUT /gb/_mapping
```

```
{
```

```
  "properties" : {
```

```
    "tag" : {
```

```
      "type" : "keyword"
```

```
    }
```

```
  }
```

```
}
```



# Double mapping

---

```
"tweet": {  
  "type": "text",  
  "analyzer": "english",  
  "fields": {  
    "fr": {  
      "type": "text",  
      "analyzer": "french"  
    }  
  }  
}
```

2 fields are available for full-text searching : *tweet* and *tweet.fr*



# *copy\_to*

The ***copy\_to*** field allows you to copy the value of a field into another field, so you can concatenate several fields into a single one

```
PUT my_index
{
  "mappings": {
    "_doc": {
      "properties": {
        "first_name": {
          "type": "text",
          "copy_to": "full_name"
        },
        "last_name": {
          "type": "text",
          "copy_to": "full_name"
        },
        "full_name": {
          "type": "text"
        }
      }
    }
  }
}
```



# Index configuration

---

Supported Data types

Text Analyzers

How to control mapping?

**Tuning Analyzers and filters**

Reindexing



# Custom analyzers

---

It is possible to define yours custom analyzers

- Either by defining precisely the tokenizer, character filters and filters to use
- Either by overriding an existing analyzer





# Creation by overload

---

In the following example, a new *fr\_std* analyzer for the *french\_docs* index is created. It uses the standard parser + the predefined list of French stopwords:

```
PUT /french_docs
{
  "settings": {
    "analysis": {
      "analyzer": {
        "fr_std": {
          "type": "standard",
          "stopwords": "_french_"
        }
      }
    }
  }
}
```



# Full creation

---

For an it is possible to define your own char filters, tokenizers, etc and use them in your own analyzers

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "char_filter": { ... custom character filters ... },
      "tokenizer": { ... custom tokenizers ... },
      "filter": { ... custom token filters ... },
      "analyzer": { ... custom analyzers ... }
    }
  }
}
```



# Example

---

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "char_filter": {
        "&_to_and": {
          "type": "mapping",
          "mappings": [ "&=> and " ]
        },
      },
      "filter": {
        "my_stopwords": {
          "type": "stop",
          "stopwords": [ "the", "a" ]
        },
      },
      "analyzer": {
        "my_analyzer": {
          "type": "custom",
          "char_filter": [ "html_strip", "&_to_and" ],
          "tokenizer": "standard",
          "filter": [ "lowercase", "my_stopwords" ]
        }
      }
    }
  }
}
```



# Using the custom analyzer

---

Once defined, the analyzer can be associated with a text field of the index

```
PUT /my_index/_mapping
{
  "properties": {
    "title": {
      "type": "text",
      "analyzer": "my_analyzer"
    }
  }
}
```



# Predefined filters

---

ELS provides many filters for building custom analyzers. For example :

- **Length Token** : Deleting words that are too short or too long
- **N-Gram** and **Edge N-Gram** : Analyzer to speed up search suggestions
- **Stemming filters** : Algorithm to extract the root of a word
- **Phonetic filters** : Phonetic representation of words
- **Synonym** : Word Match
- **Keep Word** : The opposite of stop words
- **Limit Token Count** : Limit the number of tokens associated with a document
- **Elison Token** : Handling of apostrophes (exemple : French)



# Synonym filter

---

Synonyms can be used to merge words that have nearly the same meaning.

Ex : pretty, cute, beautiful

They can also be used to make a word more generic.

- For example, bird can be used as a synonym for pigeon, sparrow, ...



# Example

---

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "filter": {
        "my_synonym_filter": {
          "type": "synonym",
          "synonyms": ["british,english", "queen,monarch"]
        }
      },
      "analyzer": {
        "my_synonyms": { "tokenizer": "standard",
                        "filter": ["lowercase","my_synonym_filter"]
        }
      }
    }
  }
}
```



# 3 syntaxes

---

Synonym replacement can be done in 3 ways:

- Simple expansion simple : If one of the terms is encountered, it is replaced by all the synonyms listed  
*"jump, leap, hop"*
- Single contraction : one of the terms encountered is replaced by a synonym  
*"leap, hop => jump"*
- Expansion générique : a term is replaced by several synonyms  
*"puppy => puppy, dog, pet"*





# Index configuration

---

Supported Data types

Text Analyzers

How to control mapping?

Tuning Analyzers and filters

**Reindexing**



# Reindexing

---

It is not possible to make some changes a posteriori to an index. (addition of analyzer for example)

To reorganize an index, the only solution is to reindex, i.e. create a new index with the new configuration and copy all documents from the old index to the new one.

The easiest way is to use the `_reindex` API



# API *\_reindex*

---

The ***\_reindex*** API consists of copying documents from one index to another index.

The target index can have a different configuration than the source index (replica, shard, mapping)

```
POST _reindex
{
  "source": {
    "index": "twitter"
  },
  "dest": {
    "index": "new_twitter"
  }
}
```



# API *reindex*

It is possible to filter the copied documents and even to query a remote cluster.

```
POST _reindex
{
  "source": {
    "remote": {
      "host": "http://otherhost:9200", // Cluster distant
      "socket_timeout": "1m",
      "connect_timeout": "10s"
    },
    "index": "source", // Sélection de l'index
    "size": 10000, // limitation sur la taille
    "query": {
      "match": { // limitation par une query
        "test": "data"
      }
    }
  },
  "dest": {
    "index": "dest"
  }
}
```



# Query syntax

---

## **DSL syntax**

Main operators

Sorting and relevance score

Advanced search

Highlighting

Aggregations or faceting

Geo queries



# Introduction

---

Searches with a request body offer more functionalities than search lite.

In particular, they allow you to :

- Combine query clauses more easily
- To influence the score
- Highlight parts of the result
- Aggregate subsets of results
- Return suggestions to the user



# GET or POST

---

RFC 7231 which deals with HTTP semantics does not define GET requests with a body

=> Only some HTTP servers support it

ELS however prefers to use the GET verb as it better describes the document retrieval action

However, so that any type of HTTP server can be put in front of ELS, GET requests with a body can also be made with the POST verb



# Empty criteria search

---

```
GET /_search
```

```
{}
```

```
GET /index_2014*/type1,type2/_search
```

```
{}
```

```
GET /_search
```

```
{
```

```
  "from": 30,
```

```
  "size": 10
```

```
}
```





# Query object

---

To use DSL, you must pass a ***query*** object in the body of the request:

```
GET /_search
```

```
{ "query": YOUR_QUERY_HERE }
```



# Structure of *query*

---

```
{  
  QUERY_NAME/OPERATOR: {  
    ARGUMENT: VALUE,  
    ARGUMENT: VALUE, ...  
  }  
}
```

Example :

GET /\_search

```
{  
  "query": {  
    "match": { "tweet": "elasticsearch" }  
  }  
}
```



# DSL Principles

---

DSL can be seen as a syntax tree that contains:

- **Leaf** query clauses.  
They correspond to a type of query (*match*, *term*, *range*) applying to a field. They can run on their own
- **Composite** clauses.  
They combine other clauses (leaf or composite) with logical operators (*bool*, *dis\_max*) or alter their behaviors (*constant\_score*)

Also, clauses can be used in 2 different contexts which modify their behavior

- **Query** or **full-text** context
- **Filter** context



# Combination example

---

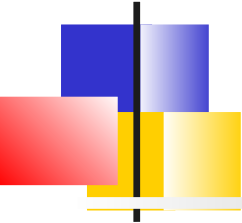
```
{  
  "bool": {  
    "must": { "match": { "tweet": "elasticsearch" } },  
    "must_not": { "match": { "name": "mary" } },  
    "should": { "match": { "tweet": "full text" } }  
  }  
}
```



# Ex: Combination of combinations

---

```
{
  "bool": {
    "must": {
      "match": { "email": "business opportunity" }
    },
    "should": [
      { "match": { "starred": true } },
      { "bool": {
        "must": { "folder": "inbox" } },
        "must_not": { "spam": true } }
      ]
    },
    "minimum_should_match": 1
  }
}
```



# Distinction between query and filter

---

## Distinction between query and filter

- **Filters** are used for fields with exact values.  
Their result is of Boolean type, i.e. a document satisfies a filter or not
- **Queries** calculate a relevance score for each document found.  
The result is sorted by the relevance score by default



# Context activation

---

The full-text context is activated when a clause is provided as a parameter to the ***query*** keyword

The filter context is activated as soon as a clause is provided as a parameter to the ***filter*** or ***must\_not*** keyword



# Example

---

GET /\_search

```
{
  "query": { // full-text context activation
    "bool": { // The clauses bool, must and match are executed in the full-text context requête
      "must": [
        { "match": { "title": "Search" } },
        { "match": { "content": "Elasticsearch" } }
      ],
      "filter": [ // filter context activation
        { "term": { "status": "published" } }, // executed in a filter context
        { "range": { "publish_date": { "gte": "2015-01-01" } } } // filter context
      ]
    }
  }
}
```





# Performance filter/full-text

---

The output of most filters is a simple list of documents that satisfy the filter. The result can be easily cached by ELS

The searches must find the documents corresponding to the keywords and in addition calculate the relevance score.

=> Searches are therefore heavier than filters and are more difficult to cache

=> The purpose of filters is to reduce the number of documents that need to be examined by the search



# Query syntax

---

DSL syntax

## **Main operators**

Sorting and relevance score

Advanced search

Highlighting

Aggregations or faceting

Geo queries



# Combination operator

---

***bool*** : Allows you to combine clauses with:

- ***must*** : equivalent to AND with score contribution
- ***filter*** : idem but does not contribute to the score
- ***must\_not*** : equivalent to NOT
- ***should*** : equivalent to OU



# Types of operators

---

Operators can be classified in 2 families :

- Operators that do not perform analysis and operate on a single term (*term*, *fuzzy*).
- The operators that apply the analyzer to the search terms corresponding to the searched field (*match*, *query\_string*) .



# Operators without analysis

---

***term*** : Used to filter exact values :

```
{ "term": { "age": 26 } }
```

***terms*** : Allows multiple values to be specified :

```
{ "terms": { "tag": [ "search", "full_text",  
"nosql" ] } }
```

***range*** : Allows you to specify a date or numeric range:

```
{ "range": { "age": { "gte": 20, "lt":  
30 } } }
```

***exists*** and ***missing*** : Allows you to test whether or not a document contains a field

```
{ "exists": { "field": "title" } }
```



## *match\_all, match\_none*

---

The ***match\_all*** search returns all documents. This is the default search if no search is specified. All documents are considered equally relevant, they receive a `_score` of 1.

```
{ "match_all": {}}
```

The opposite of *match\_all* is ***match\_none*** which returns no documents.

```
{ "match_none": { }}
```



# *match*

---

The match ***operator*** is the standard operator for performing an exact or full-text search on almost any field.

- If the query is for a full-text field, it parses the search string using the same parser as the field,
- If the search is for an exact value field, it searches for the exact value

```
{ "match": { "tweet": "About Search" } }
```



# OR by default

---

*match* is a Boolean query which by default analyzes the words passed as parameters and builds an OR type query.

*match* applies on **a field**

It can also precises :

- ***operator*** (and/or) :
- ***minimum\_should\_match*** : the number of clauses to match
- ***analyzer*** : the parser to use for the search string
- ...





# Example

---

GET /\_search

```
{  
  "query": {  
    "match" : {  
      "message" : {  
        "query" : "this is a test",  
        "operator" : "and"  
      }  
    }  
  }  
}
```



# Control combination

---

```
GET /my_index/my_type/_search
{
  "query": { "match": {
    "title": {
      "query": "quick brown fat dog",
      "minimum_should_match": "75%" }
    } } }
```

Documents containing 75% of the specified words



# *multi\_match*

---

The ***multi\_match*** operator allows you to run the same query on multiple fields:

```
{"multi_match": { "query": "full text search", "fields":  
[ "title", "body" ] } }
```

Wildcards can be used for fields

Fields can be boosted with notation ^

```
GET /_search  
{  
  "query": {  
    "multi_match" : {  
      "query" : "this is a test",  
      "fields" : [ "subject^3", "text*" ]  
    }  
  }  
}
```



# *query\_string*

---

***query\_string*** operator uses a parser to understand the query string. (the same as for lite search)

It has many parameters (default\_field, analyzer, ...)

```
GET /_search
{
  "query": {
    "query_string" : {
      "default_field" : "content",
      "query" : "title:(quick OR brown)"
    }
  }
}
```



## *simple\_query\_string*

---

The ***simple\_query\_string*** operator avoids causing a parsing error in the search query.

It can be used directly with what a user has entered in a search field

It understands the simplified syntax:

- + for AND
- | for OR
- - for negation
- ...



# Validation of queries

---

```
GET /gb/tweet/_validate/query?explain
{ "query": { "tweet" : { "match" : "really powerful" } }
...
{
  "valid" : false,
  "_shards" : { ... },
  "explanations" : [ {
    "index" : "gb",
    "valid" : false,
    "error" : "org.elasticsearch.index.query.QueryParseException:
[gb] No query registered for [tweet]"
  } ]
}
```



# Query syntax

---

DSL syntax

Main operators

**Sorting and relevance score**

Advanced search

Highlighting

Aggregations or faceting

Geo queries



# Sorting

---

During filter-type queries, it may be interesting to set the sorting criterion.

This is done via the **sort** parameter

GET /\_search

```
{
  "query" : {
    "bool" : {
      "filter" : { "term" : { "user_id" : 1 } }
    }
  }, "sort": { "date": { "order": "desc" } }
}
```





# Response

In the response, the `_score` field is not calculated and the value of the `sort` field is specified for each document

```
"hits" : {
  "total" : 6,
  "max_score" : null,
  "hits" : [ {
    "_index" : "us",
    "_type" : "tweet",
    "_id" : "14",
    "_score" : null,
    "_source" : {
      "date": "2014-09-24",
      ...
    }, "sort" : [ 1411516800000 ]
  },
  ...
}
```



# Several sorting criteria

---

It is possible to combine a sorting criterion with another criterion or the score. Please note the order is important.

```
GET /_search
{
  "query" : {
    "bool" : {
      "must": { "match": { "tweet": "manage text search" }},
      "filter" : { "term" : { "user_id" : 2 }}
    }
  }, "sort": [
    { "date":
      { "order": "desc" }},
    { "_score": { "order": "desc" }}]
}
```



# Sorting on multivalued fields

---

It is also possible to use an aggregate function to sort multivalued fields (*min*, *max*, *sum*, *avg*, ...)

```
"sort": {  
  "dates": {  
    "order": "asc",  
    "mode": "min"  
  }  
}
```



# Relevance

---

The score of each document is represented by a floating point number (`_score`). The higher this number, the more relevant is the document.

The algorithm used by ELS is called ***term frequency/inverse document frequency***, or ***TF/IDF***.

It takes into account the following factors:

- The ***frequency of the term*** : how many times the term appears in the field
- The ***inverse document frequency*** : How many times does the term appear in the index? The more the term appears, the less weight it has.
- The ***normalized length of the field*** : The longer the field, the less relevant the terms

Depending on the type of query (fuzzy query, boost factor ...) other factors can influence the score



# Explanation of relevance

---

With the ***explain*** parameter, ELS provides an explanation of the score

```
GET /_search?explain
```

```
{ "query"
: { "match" : { "tweet" : "honeymoon" }}
}
```

The *explain* parameter can also be used to understand why a document matches or not.

```
GET /us/12/_explain
```

```
{
"query" : {
  "bool" : {
    "filter" : { "term" : { "user_id" : 2 }},
    "must" : { "match" : { "tweet" : "honeymoon" }}
  } } }
```



# Response

---

```
"_explanation": {
  "description": "weight(tweet:honeyymoon in 0)
[PerFieldSimilarity], result of:",
  "value": 0.076713204,
  "details": [ {
    "description": "fieldWeight in 0, product of:",
    "value": 0.076713204,
    "details": [ {
      "description": "tf(freq=1.0), with freq of:",
      "value": 1,
      "details": [ {
        "description": "termFreq=1.0",
        "value": 1
      } ]
    }, {
      "description": "idf(docFreq=1, maxDocs=1)",
      "value": 0.30685282
    }, {
      "description": "fieldNorm(doc=0)",
      "value": 0.25,
    } ]
  } ]
} ] }
```



# *multi\_match*

---

The ***multi\_match*** query makes it easy to run the same query on multiple fields.

Wildcard can be used as well as individual boost

```
{  
  "multi_match": {  
    "query": "Quick brown fox",  
    "type": "best_fields",  
    "fields": [ "*_title^2", "body" ],  
    "tie_breaker": 0.3,  
    "minimum_should_match": "30%"  
  }  
}
```



# *multi\_match*

---

With the operator *multi\_match*, the ***type*** parameter specifies the way the score is calculated.

- ***best\_fields*** (default) : Find documents that match one of the fields but use the best field to assign the score
- ***most\_fields*** : Combines the score of each field
- ***cross\_fields*** : Concatenates all fields and uses the same parser
- ***phrase*** : Use a *match\_phrase* operator on each field and combine each field's score
- ***phrase\_prefix*** : Use a *match\_phrase\_prefix* operator on each field and combine each field's score





# bool operator

---

```
GET /my_index/my_type/_search
{ "query": {
  "bool": {
    "must": { "match": { "title": "quick" }},
    "must_not": { "match": { "title": "lazy" }},
    "should": [
      { "match": { "title": "brown" }},
      { "match": { "title": "dog" }}
    ]
  }
}
```

The calculation of relevance is done by adding the score of each *must* or *should* clause and dividing by 3



# Boosting clause

---

It is possible to give more weight to a particular clause by using the **boost** parameter

```
GET /_search
{
  "query": { "bool": {
    "must": { "match": {
      "content": { "query": "full text search", "operator": "and" }}}},
    "should": { "match": {
      "content": { "query": "Elasticsearch", "boost": 3 }}}
  } } }
```



# *boosting* operator

The ***boosting*** operator allows you to specify a clause that reduces the score of matching documents

```
GET /_search
```

```
{
  "query": {
    "boosting" : {
      "positive" : {
        "term" : {"text" : "apple"}
      },
      "negative" : {
        "term" : { "text" : "pie tart fruit crumble tree" }
      },
      "negative_boost" : 0.5 // requis le malus au document qui matche
    }
  }
}
```



# *dis\_max*

Instead of combining queries via `bool`, it may be more relevant to use ***dis\_max***

*dis\_max* is an OR but relevance calculation differs

*Disjunction Max Query* means: return the documents that match one of the queries and return the score of the best matching query

```
{ "query": { "dis_max": {  
  "queries": [  
    { "match": { "title": "Brown fox" }},  
    { "match": { "body": "Brown fox" }}  
  ]  
} } }
```



# *tie\_breaker*

---

It is also possible to take into account other queries that match by giving them a lower importance.

This is the ***tie\_breaker*** parameter

```
{
  "query": {
    "dis_max": { "queries": [
      { "match": { "title": "Quick pets" }},
      { "match": { "body": "Quick pets" }}
    ], "tie_breaker": 0.3 }
  } }
```

The score calculation is then performed as follows:

- 1. Take the score of the best clause.
- 2. Multiply the score of each other matching clause by the tie\_breaker.
- 3. Add and normalize them



# Query syntax

---

DSL syntax  
Main operators  
Sorting and relevance score  
**Advanced search**  
Highlighting  
Aggregations or faceting  
Geo queries



# Partial Matching

---

Partial matching allows users to specify a portion of the term they are looking for

Common use cases are:

- Matching postal codes, serial numbers or other not\_analyzed values that start with a particular prefix or even a regular expression
- On-the-fly search: searches performed on each character typed to make suggestions to the user
- Matching in languages like German that contain long compound nouns



# *prefix* filter

---

The *prefix* filter is a search performed on the term. It does not parse the search string and assumes that the correct prefix has been provided.

```
GET /my_index/address/_search
{
  "query": {
    "prefix": { "postcode": "W1" }
  }
}
```





# Wildcard and regexp

**wildcard** or **regexp** searches are similar to *prefix* but allow the use of wildcards or regular expressions

```
GET /my_index/address/_search
{
  "query": {
    "wildcard": { "postcode": "W?F*HW" }
  }
}
```

```
GET /my_index/address/_search
{
  "query": {
    "regexp": { "postcode": "W[0-9].+" }
  }
}
```



# Indexing for auto-completion

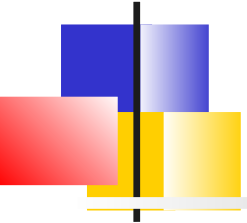
---

The principle is to index the beginnings of words of each term

This can be done through a particular ***edge\_ngram*** filter

```
{  
  "filter": {  
    "autocomplete_filter": { "type": "edge_ngram",  
                             "min_gram": 1,  
                             "max_gram": 20  
    }  
  }  
}
```

For each term, it creates n tokens of minimum size 1 and maximum 20. The tokens are the different prefixes of the term.

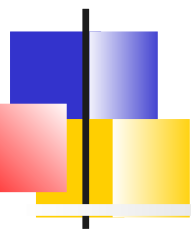


# *match\_phrase*

---

The ***match\_phrase*** operator parses the search string to produce a list of terms but only keeps documents that contain all of the terms in the same position.

```
GET /my_index/my_type/_search
{
  "query": {
    "match_phrase": { "title": "quick brown
fox" }
  }
}
```



# Proximity and *slop* parameter

---

It is possible to introduce flexibility to phrase matching by using the *slop* parameter which indicates how far apart the terms can be.

Documents with these closest terms will have better relevance.

```
GET /my_index/my_type/_search
{
  "query": {
    "match_phrase": {
      "title": {
        "query": "quick fox",
        "slop": 20 // ~ distance in words
      }
    }
  }
}
```



# *match\_phrase\_prefix*

The ***match\_phrase\_prefix*** operator behaves like *match\_phrase*, except it treats the last word as a prefix

It is possible to limit the number of expansions by setting the *max\_expansions* parameter

```
{  
  "match_phrase_prefix" : {  
    "brand" : {  
      "query": "johnnie walker bl",  
      "max_expansions": 50  
    }  
  }  
}
```



# Fuzzy query

---

**Fuzzy query** returns documents that contain terms similar to the search term, as measured by a Levenshtein edit distance.

An edit distance is the number of one-character changes needed to turn one term into another. These changes can include:

- Changing a character (box → fox)
- Removing a character (black → lack)
- Inserting a character (sic → sick)
- Transposing two adjacent characters (act → cat)

Most typos or misspelling errors have a distance of 1.

=> So 80% of errors could be fixed with single character editing



# Fuzzy query

---

It is possible during a request to position the **fuzziness** parameter which therefore gives the Levenshtein distance tolerance.

This parameter can also be set to AUTO:

- 0 for strings of 1 or 2 characters
- 1 for strings of 3, 4 or 5 characters
- 2 for strings longer than 5 characters

Be careful: No stemmer with fuzzy queries!



# Example

---

```
GET /my_index/my_type/_search
```

```
{  
  "query": {  
    "multi_match": {  
      "fields": [ "text", "title" ],  
      "query": "SURPRIZE ME!",  
      "fuzziness": "AUTO"  
    }  
  }  
}
```





# *fuzzy* operator

---

The fuzzy search is equivalent to a search by term

```
GET /my_index/my_type/_search
{
  "query": {
    "fuzzy": {"text": "surprise" }
  }
}
```

2 parameters can be used to limit the performance impact:

- ***prefix\_length*** : The number of initial characters that will not be changed.
- ***max\_expansions*** : Limit the options that fuzzy search generates. Fuzzy search stops gathering nearby terms when it reaches this limit



# Query syntax

---

DSL syntax  
Main operators  
Sorting and relevance score  
Advanced search  
**Highlighting**  
Aggregations or faceting  
Geo queries



# Introduction

---

**Highlighting** consists of highlighting the matched term in the original content that has been indexed.

This is possible thanks to the metadata stored during indexing

The original field must be stored by ELS



# Example

---

```
GET /_search
{
  "query" : {
    "bool" : {
      "must" : { "match" :
{ "attachment.content" : "Administration"} },
      "should" : { "match" : { "attachment.content" :
"Oracle" } }
    }
  },
  "highlight" : {
    "fields" : {
      "attachment.content" : {}
    }
  }
}
```



# Response

---

```
"highlight" : {  
    "attachment.content" : [  
        " formation <em>Administration</em> <em>Oracle</em>  
Forms/Report permet de voir tous les aspects nécessaires à",  
        " aux autres services Web de l'offre  
<em>Oracle</em> Fusion Middleware. Cependant, l'administration  
de ces",  
        " Surveillance. Elle peut-être un complément de la  
formation « <em>Administration</em> d'un serveur Weblogic",  
        "-forme <em>Oracle</em> Forms ou des personnes  
désirant migrer leur application client serveur Forms vers",  
        " <em>Oracle</em> Forms 11g/12c\nPré-requis : \  
nExpérience de l'administration système\nLa formation  
« Administation"  
    ]  
}
```



# Tags used in the response

---

```
GET /_search
{
  "query" : {
    "bool" : {
      "must" : { "match" :
{ "attachment.content" : "Administration" } },
      "should" : { "match" : { "attachment.content" : "Oracle" } }
    }
  },
  "highlight" : {
    "pre_tags" : ["<tag1>"],
    "post_tags" : ["</tag1>"],
    "fields" : {
      "attachment.content" : { }
    }
  }
}
```



# Fragments

---

It is possible to control the highlighted fragments for each field:

- ***fragment\_size*** : gives the max size of the highlighted fragment
- ***number\_of\_fragments*** : The maximum number of fragments in this field



# Query syntax

---

DSL syntax  
Main operators  
Sorting and relevance score  
Advanced search  
Highlighting  
**Aggregations or faceting**  
Geo queries





# Introduction

---

Aggregations are extremely powerful for reporting and dashboards

Using the Elastic, Logstash, and Kibana stack demonstrates just how much you can do with aggregations.

# Kibana dashboard





# DSL Syntax

---

An aggregation can be seen as a unit of work that builds analytical information on a set of documents.

Depending on its position in the DSL tree, it applies to all search results or to subsets

In DSL syntax, an aggregation block uses the keyword ***aggs***

**// The max of the price field in all documents**

```
POST /sales/_search?size=0
```

```
{
  "aggs" : {
    "max_price" : { "max" : { "field" : "price" } }
  }
}
```



# Types of aggregations

---

Several concepts relate to aggregations:

- **Groups or Buckets** : Set of documents that have a field with the same value or share the same criteria.  
Groups can be nested. ELS provides syntaxes for defining groups and counting the number of documents in each category
- **Metrics**: Metric calculations on a group of documents (min, max, avg, ..)
- **Pipeline** : Aggregations that take input from other aggregations instead of documents or fields.



# Example Bucket

---

GET /cars/transactions/\_search

```
{  
  "aggs" : {  
    "colors" : {  
      "terms" : { "field" : "color.keyword" }  
    } } }
```



# Response

---

```
{  
  ...  
  "hits": { "hits": [] },  
  "aggregations": {  
    "colors": {  
      "doc_count_error_upper_bound": 0, // incertitude  
      "sum_other_doc_count": 0,  
      "buckets": [  
        { "key": "red", "doc_count": 4 },  
        { "key": "blue", "doc_count": 2 },  
        { "key": "green", "doc_count": 2 }  
      ]  
    }  
  }  
}
```



# Example metric

---

GET /cars/transactions/\_search

```
{
  "size": 0,
  "aggs" : {
    "avg_price" : {
      "avg" : {"field" : "price"}
    }
  }
}
```



# Response

---

```
"hits": {  
  "total": 8,  
  "max_score": 0,  
  "hits": []  
},  
"aggregations": {  
  "avg_price": {  
    "value": 26500  
  }  
}
```





# Bucket/Metrics juxtaposition

---

```
GET /cars/transactions/_search
```

```
{
  "size": 0,
  "aggs" : {
    "colors" : {
      "terms" : { "field" : "color.keyword" }
    },
    "avg_price" : {
      "avg" : {"field" : "price"}
    }
  }
}
```



# Response

---

```
{  
  ...  
  "hits": { "hits": [] },  
  "aggregations": {  
    "avg_price": {  
      "value": 26500  
    },  
    "colors": {  
      "doc_count_error_upper_bound": 0,  
      "sum_other_doc_count": 0,  
      "buckets": [  
        { "key": "red", "doc_count": 4 },  
        { "key": "blue", "doc_count": 2 },  
        { "key": "green", "doc_count": 2 }  
      ]  
    }  
  }  
}}
```



# Nesting aggregations

---

```
GET /cars/transactions/_search {  
  "aggs": {  
    "colors": {  
      "terms": { "field": "color" },  
      "aggs": {  
        "avg_price": {  
          "avg": { "field": "price" }  
        }, "make": {  
          "terms": { "field": "make" }  
        }  
      }  
    }  
  }  
}
```



# Response

---

```
{
  ...
  "aggregations": {
    "colors": {
      "buckets": [
        { "key": "red",
          "doc_count": 4,
          "avg_price": {
            "value": 32500
          },
          "make": { "buckets": [
            { "key": "honda", "doc_count": 3 },
            { "key": "bmw", "doc_count": 1 }
          ]
        }
      ],
    },
    ...
  }
}
```



# Aggregation and search

---

In general, an aggregation is combined with a search. The buckets are then deduced from the documents that match.

```
GET /cars/transactions/_search
{
  "query" : {
    "match" : { "make" : "ford" }
  }, "aggs" : {
    "colors" : {
      "terms" : { "field" : "color" }
    }
  }
}
```



# Sorting buckets

---

GET /cars/transactions/\_search

```
{
  "aggs" : {
    "colors" : {
      "terms" : { "field" : "color",
                  "order": { "avg_price" : "asc" }
                },
      "aggs": {
        "avg_price": {
          "avg": {"field": "price"}
        }
      }
    }
  }
}
```



# Pipeline

---

Pipeline aggregations work on the outputs produced from other aggregations

They reference the aggregations used to perform their computation by using the ***buckets\_path*** parameter



# Example

---

POST /\_search

```
{
  "aggs": {
    "my_date_histo": {
      "date_histogram": {
        "field": "timestamp",
        "calendar_interval": "day"
      },
      "aggs": {
        "the_sum": {
          "sum": { "field": "lemmings" }
        },
        "the_deriv": {
          "derivative": { "buckets_path": "the_sum" }
        }
      }
    }
  }
}
```





# Types of buckets

---

ELS offers different ways of grouping data :

- By term: Requires field tokenization
- Histogram : By range of values
- Date Histogram : By date range
- By IP range
- By absence/presence of a field
- Significant terms
- By geo-location
- ..



# Histogram

---

```
GET /cars/transactions/_search
{
  "aggs": {
    "price": {
      "histogram": {
        "field": "price",
        "interval": 20000
      },
      "aggs": {
        "revenue": {
          "sum": { "field" : "price" }
        }
      }
    }
  }
}
```



# Date histogram

---

```
GET /cars/transactions/_search
```

```
{
```

```
  "aggs": {
```

```
    "sales": {
```

```
      "date_histogram": {
```

```
        "field": "sold",
```

```
        "interval": "month",
```

```
        "format": "yyyy-MM-dd"
```

```
      }
```

```
    } } }
```



# *significant\_terms*

---

The ***significant\_terms*** aggregation is more subtle but can give interesting results (anomaly detection).

This consists of analyzing the returned data and finding the terms that appear at an abnormally higher frequency

Abnormally means: relative to the frequency for all documents

=> These statistical anomalies generally reveal interesting things

# Mechanism



*significant\_terms* takes a search result and performs an aggregation

It then starts from the set of documents and performs the same aggregation

It then compares the results of the first search that are "abnormal" against the overall search

With this type of operation, you can:

- People who liked... also liked...
- Customers who had questionable credit card transactions all went to such and such a merchant
- Every Thursday evening, the page is much more consulted



# Example

---

```
{  
  "query" : {  
    "terms" : { "force" : [ "British Transport Police" ] }  
  },  
  "aggregations" : {  
    "significantCrimeTypes" : {  
      "significant_terms" : { "field" : "crime_type" }  
    }  
  }  
}
```



# Response

---

```
"aggregations" : {  
  "significantCrimeTypes" : {  
    "doc_count": 47347, // Total query result  
    "buckets" : [  
      {  
        "key": "Bicycle theft",  
        "doc_count": 3640, // Number docs for the query result  
        "score": 0.371235374214817,  
        "bg_count": 66799 // Number for all documents  
      }  
      ...  
    ]  
  }  
}
```

=> Bike theft rate unusually high for « British Transport Police »



# Available metrics

---

ELS offers many metrics:

- ***avg, min, max, sum***
- ***value\_count, cardinality*** : Distinct value count
- ***top\_hit*** : The top documents
- ***extended\_stats*** : Statistical metrics (count, sum, variance, ...)
- ***percentiles*** : percentiles





# Query syntax

---

DSL syntax  
Main operators  
Sorting and relevance score  
Advanced search  
Highlighting  
Aggregations or faceting  
**Geo queries**



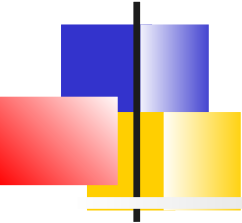
# Introduction

---

ELS allows you to combine geo-location with full-text, structured searches and aggregations

ELS has 2 data types to represent geolocation data

- ***geo\_point*** : represents a latitude-longitude couple. This mainly allows the calculation of distance
- ***geo\_shape*** : defines an area. This allows you to know if 2 areas have an intersection



# Geo-point

Geo-points cannot be automatically detected by ELS. They must be explicitly specified in the mapping:

```
PUT /attractions
```

```
{ "mappings": { "restaurant": {  
  "properties": {  
    "name": { "type": "string" },  
    "location": { "type": "geo_point" }  
  }  
} } }
```

----

```
PUT /attractions/restaurant/1
```

```
{"name": "Chipotle Mexican Grill", "location": "40.715, -74.011" }
```

```
PUT /attractions/restaurant/2
```

```
{ "name": "Pala Pizza", "location": { "lat":40.722,"lon": -73.989 } }
```

```
PUT /attractions/restaurant/3
```

```
{ "name": "Mini Munchies Pizza","location": [ -73.983, 40.719 ] }
```



# Filters

---

4 filters can be used to include or exclude documents with respect to their *geo-point*:

- ***geo\_bounding\_box***: The geo-points included in the provided rectangle
- ***geo\_distance***: Distance from a center point below a boundary.  
Sorting and scoring can be relative to distance
- ***geo\_distance\_range***: Distance in a range
- ***geo\_polygon***: Geo-points include in a polygon



# Example

---

GET /attractions/restaurant/\_search

```
{  
  "query": {  
    "bool": {  
      "filter": {  
        "geo_bounding_box": {  
          "location": { "top_left": { "lat": 40.8, "lon": -74.0 },  
                        "bottom_right": { "lat": 40.7, "lon": -73.0 }  
        }  
      }  
    }  
  }  
}
```



# Aggregation

---

3 types of aggregation on geo-points are possible

- ***geo\_distance*** (bucket): Groups documents in concentric circles around a central point
- ***geohash\_grid*** (bucket): Group documents by cells (*geohash\_cell*, google maps squares) for display on a map
- ***geo\_bounds*** (metrics): returns the coordinates of a rectangle area that would encompass all geo-points. Useful for choosing the right zoom level



# Example

---

```
GET /attractions/restaurant/_search
{
  "query": { "bool": { "must": {
    "match": { "name": "pizza" }
  },
  "filter": { "geo_bounding_box": {
    "location": { "top_left": { "lat": 40.8, "lon": -74.1 },
                  "bottom_right": { "lat": 40.4, "lon": -73.7 }
    }
  } } } },
  "aggs": {
    "per_ring": {
      "geo_distance": {
        "field": "location",
        "unit": "km",
        "origin": {
          "lat": 40.712,
          "lon": -73.988
        },
        "ranges": [
          { "from": 0, "to": 1 },
          { "from": 1, "to": 2 }
        ]
      }
    }
  }
}
```



# Geo-shape

---

Like *geo\_point* , ***geo-shape*** fields must be mapped explicitly:

```
PUT /attractions
```

```
{  
  "mappings": { "landmark": {  
    "properties": {  
      "name": { "type": "string" },  
      "location": { "type": "geo_shape" }  
    } } } }  
}
```

```
---
```

```
PUT /attractions/landmark/dam_square
```

```
{  
  "name" : "Dam Square, Amsterdam",  
  "location" : {  
    "type" : "polygon",  
    "coordinates" : [[ [ 4.89218, 52.37356 ], [ 4.89205, 52.37276 ], [ 4.89301, 52.37274 ],  
    [ 4.89392, 52.37250 ], [ 4.89218, 52.37356 ] ]  
  } }  
}
```





# Query example

---

```
GET /attractions/landmark/_search
{
  "query": {
    "geo_shape": {
      "location": {
        "shape": {
          "type": "circle",
          "radius": "1km"
          "coordinates": [ 4.89994, 52.37815]
        }
      }
    }
  }
}
```



# Elastic Search Clients

---

## **Clients libraries**

Kibana's vizualisation and dashboards



# Introduction

---

ElasticSearch distributes a set of integration libraries for different languages or frameworks:

- Java + framework SpringBoot + ...
- JavaScript
- Ruby
- Go
- .NET
- PHP
- Perl
- Python

Other libraries provided by third-parties are also available



# Features

The use of a library (comparer to low level REST calls) brings some benefits :

- Strongly typed requests and responses for all Elasticsearch APIs.
- Blocking and asynchronous versions of all APIs.
- Use of fluent builders and functional patterns to allow writing concise yet readable code when creating complex nested structures.
- Seamless integration of application classes by using an object mapper such as Jackson or any JSON-B implementation.
- Delegates protocol handling to an http client such as the Java Low Level REST Client that takes care of all transport-level concerns: HTTP connection pooling, retries, node discovery, and so on.



# Example Java : Connection

---

```
// Create the low-level client
RestClient restClient = RestClient.builder(
    new HttpHost("localhost", 9200)).build();

// Create the transport with a Jackson mapper
ElasticsearchTransport transport = new
RestClientTransport(
    restClient, new JacksonJsonpMapper());

// And create the API client
ElasticsearchClient client = new
ElasticsearchClient(transport);
```



# Request

---

```
SearchResponse<Product> search = client.search(s -> s
    .index("products")
    .query(q -> q
        .term(t -> t
            .field("name")
            .value(v -> v.stringValue("bicycle")))
        )), Product.class);

for (Hit<Product> hit: search.hits().hits()) {
    processProduct(hit.source());
}
```



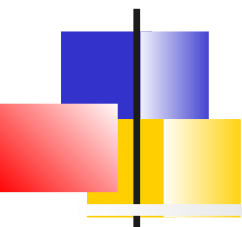
# Elastic Search Clients

---

Clients libraries

**Kibana's visualisation and  
dashboards**

# Introduction



---

Kibana is an analytics and visualization platform powered by Elasticsearch

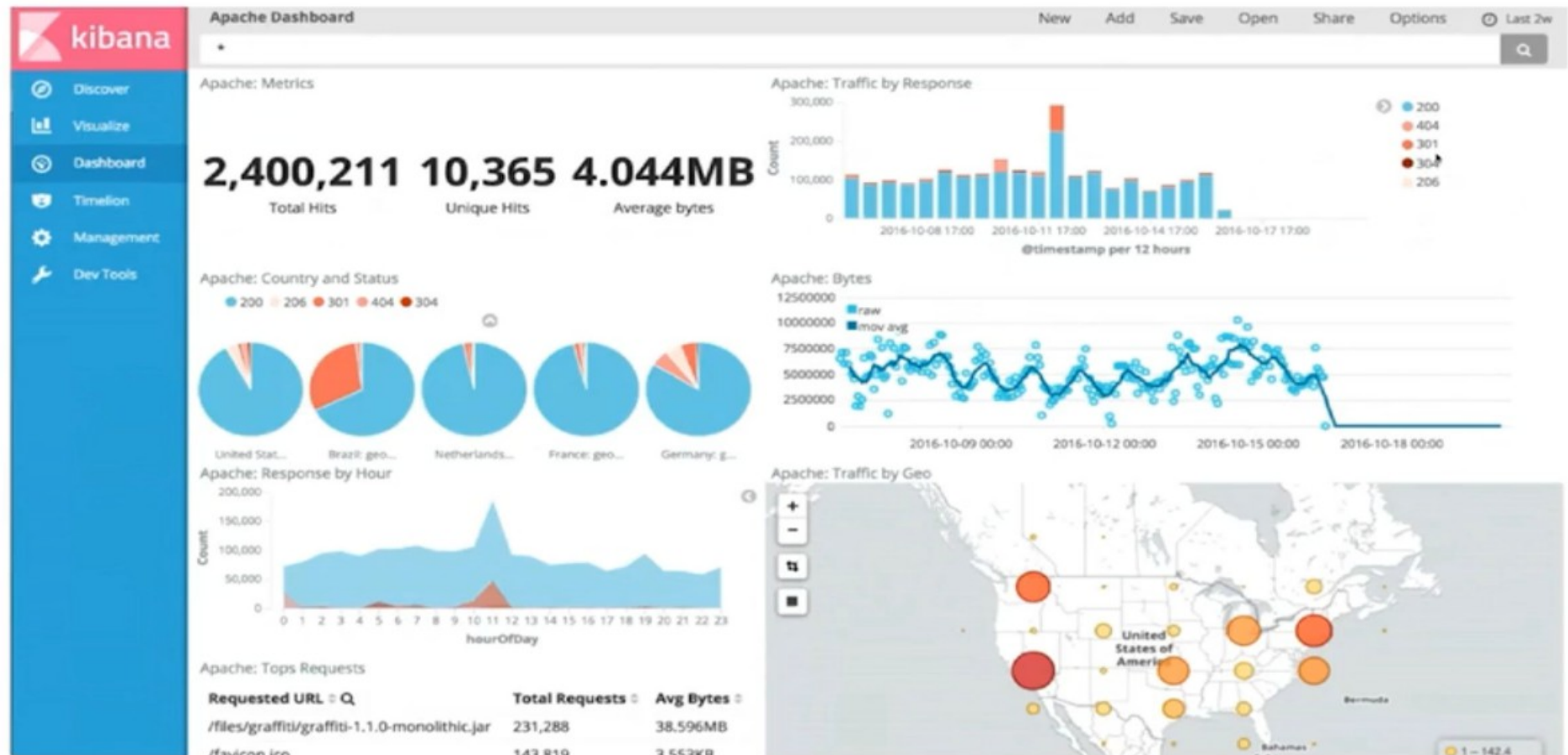
It is able to search data stored in Elasticsearch indexes

It offers a web interface for creating dynamic dashboards displaying the results of queries in real time

It runs on Node.js



# Dashboard Kibana





# Final users and use cases

---

Kibana to destinations of all profiles: Business, Operations, Developer

Once data is ingested into Elasticsearch indexes, it allows:

- Explore the data to better understand it
- Create visualizations and include them in dashboards
- Create presentations to facilitate meetings
- Share via email, web page or PDF documents
- Apply Machine Learning models to detect anomalies or anticipate the future
- Generate graphs visualizing the relationships between your data
- Manage Elasticsearch indexes
- Generate alerts about your data
- Organize Kibana assets into "Space" related to ES security and permissions model



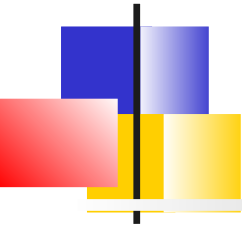
# Analyze and visualize your data

---

In the context of the Enterprise Search features of ELK, Kibana can be used :

- Adjust our ELS queries with the **Dev Console**
- Understand your Data with the menu **Discover**
- Analyze your data with **Visualizations** and **Dashboards**

# Steps of developing a dashboard



Defining an Index Pattern and its timestamp field

Explore data and optionally save queries

Create visualizations, aggregations, times, maps

Arrange them on a dashboard

Share Dashboard URL



# Index Pattern creation

---

To exploit data, it is necessary to define ***index pattern***, i.e. collection of indexes respecting a naming rule:

- ex: logstash-\*
- You must also indicate the *timestamp* field of these indexes

Then, Kibana loads the mapping information of these indexes and clearly indicates their type, whether they can be used for search or for aggregation.

# Discover Menu

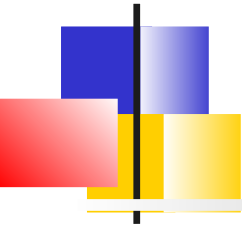
The image shows the Kibana Discover interface with several components labeled:

- Index Pattern:** Points to the `logstash-*` dropdown menu.
- Query bar:** Points to the search bar containing an asterisk `*`.
- Time Picker:** Points to the time range selector showing `May 17th 2015, 04:00:41.685 to May 20th 2015, 18:32:51.964`.
- Toolbar:** Points to the top right area containing buttons for `New`, `Save`, `Open`, `Share`, and a search icon.
- Histogram:** Points to the bar chart showing the distribution of data over time, with the x-axis labeled `utc_time per hour` and the y-axis labeled `Count`.
- Document Table:** Points to the table of search results below the histogram.
- Side Navigation:** Points to the left sidebar menu containing `Discover`, `Visualize`, `Dashboard`, `Timelion`, `Management`, and `Dev Tools`.

The **Document Table** displays the following data:

| Time                        | _source  |
|-----------------------------|--|
| May 18th 2015, 02:03:25.877 | <code>@timestamp: May 18th 2015, 02:03:25.877 ip: 185.124.182.126 extension: gif response: 404 geo.coordinates: { "lat": 36.518375, "lon": -86.05828083 } geo.src: PH geo.dest: MM geo.srcdest: PH:MM @tags: success, info utc_time: May 18th 2015, 02:03:25.877 referer: http://twitter.com/error/will</code> |
| May 18th 2015, 05:28:25.013 | <code>@timestamp: May 18th 2015, 05:28:25.013 ip: 79.1.14.87 extension: gif response: 200 geo.coordinates: { "lat": 35.16531472, "lon": -107.9006142 } geo.src: GN geo.dest: US geo.srcdest: GN:US @tags: success, info utc_time: May 18th 2015, 05:28:25.013 referer: http://www.slate.com/warning/</code>    |

# Time window



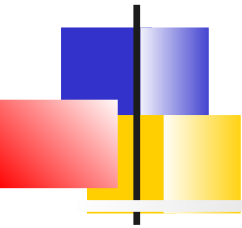
---

If a timestamp field exists in the index:

- The top of the page displays the distribution of documents over a period (by default 15 min)
- Time window can be changed

This time window is present in the majority of Kibana pages

# Search



Search criteria can be entered in the query bar. It could be:

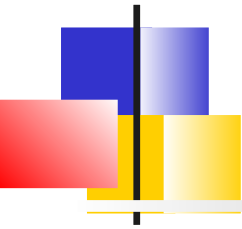
- A simple text
- A Lucene query
- A DSL request with JSON.

The result updates the whole page and only the first 500 documents are returned in reverse chronological order

- Field filters can be specified
- Search can be saved with a name
- The index pattern can be modified
- Search can automatically refresh every X times



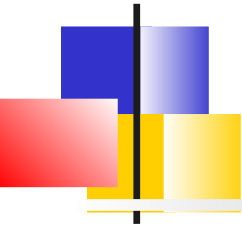
# Filters



Several buttons are available on filters:

- Activation deactivation
- Pin: Filter is retained even on a context switch
- Toggle: Positive or Negative Filter
- Deletion
- Edit: One can then work with DSL syntax and create filter combinations

# Document viewing



---

The list displays by default 500 documents  
(property discover:sampleSize)

It is possible to choose the sorting criterion

Add/Remove Fields

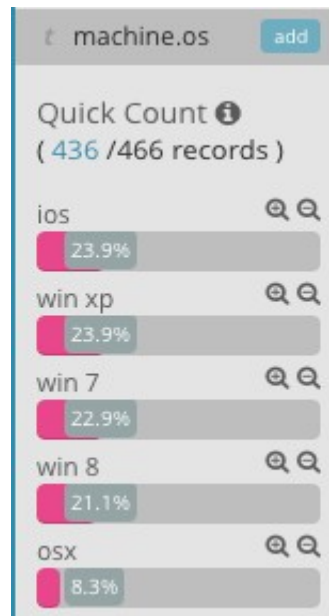
Statistics are also available



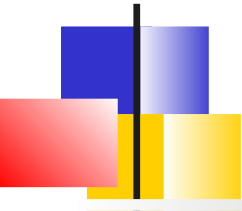
# Field value statistics

---

Field value statistics show how many documents have a particular value in a field (facetting)



# Visualizations



The visualizations are based on the aggregation capabilities of ELS, we can create graphs that show trends, peaks, ...

Visualizations can then be grouped into dashboards

Different editors for visualization are available:

- Lens: Drag and Drop
- Classical Visualizations based on aggregations
- Maps: Geolocation
- TSVB et Timelion: Suitable for time series
- Custom visualizations: Vega



# Classical visualization

---

**Area** : View total contributions from multiple series

**Table de données** : Tabular display of aggregated data

**Line** : Compare different series

**Metric**: Display a single metric

**Gauge/Goal**: A single value with ranges of good/bad values relative to a target

**Pie** : Pie .

**Heat map** : Heat map

**Nuage de tags** : The most important tags have the largest font

**Horizontal / Vertical bar** : Histogram allowing the comparison of series



# Steps of creation

---

With the classical editor steps are

1. Choose a chart type
  2. Specify search criteria or use a saved search
  3. Choose the aggregation calculation for the Y axis (count, average, sum, min, ...)
  4. Choose the data grouping criterion (bucket expression)  
(Histogram of date, Interval, Terms, Filters, Significant terms, ...)
  5. Optionally define sub-aggregations
- Lens allows you to create your visualization more intuitively via Drag & Drop

# Dashboard



---

A dashboard groups a set of saved visualizations into a web page

It is possible to rearrange and resize visualizations

It is possible to share the dashboards by a simple link