

## Elastic Stack 을 활용한 Data Dashboard 만들기

Week 4 - Elasticsearch API를 활용해보자



Fast Campus

내용	페이지
Dev Tools	3
Data Type	
Core datatype	11
Complex datatype	20
설치	39
API	
Indicies API	
Create Index	46
Delete Index	47
Mapping	48
Document API	
Create Document	61
Get Document	63
Delete Document	64
Update Document	66
Reindex Document	70
Search API (Query DSL)	
Match All	81
Term/Terms	83
Prefix/Wildcard/Fuzzy	85
Range	88
Query String	91
Exists	89
Bool	94

# Kibana Dev Tools를 살펴보자

오늘 배우는 API는 대부분 여기에 작성한다

# Kibana에 접속해서 Dev Tools 화면으로 가자



## 다음과 같이 입력하고 녹색 버튼을 눌러보자

Dev Tools

Console

History Settings Help

 2. 선택



 1. 입력

```
1 GET /shopping/_search
2 {
3   "query": {
4     "match_all": {}
5   }
6 }
```

```
1 {
2   "took": 0,
3   "timed_out": false,
4   "_shards": {
5     "total": 5,
6     "successful": 5,
7     "skipped": 0,
8     "failed": 0
9   },
10  "hits": {
11    "total": 6030,
12    "max_score": 1,
13    "hits": [
14      {
15        "_index": "shopping",
16        "_type": "shopping",
17        "_id": "AWJcYublQ1U_UGT8VEJ",
18        "_score": 1,
19        "_source": {
20          "접수 번호": 391,
21          "주문 시간": "2018-02-12T16:03:38",
22          "수령 시간": "2018-02-13T05:19:38",
23          "예약 여부": "일반",
24          "배송 메모": "시간 내에 배송 못함",
25          "고객 ip": "213.75.93.244",
26          "고객 성별": "여성",
27          "고객 나이": 25,
28          "물건 좌표": "35.34784911832476, 126.58228016535746",
29          "고객 주소_시도": "충청남도",
30          "구매 사이트": "위메프",
31          "판매자 평점": 2,
32          "상품 분류": "팬츠",
33          "상품 가격": 27000,
34          "상품 개수": 7,
35          "결제 카드": "시티"
36        }
37      },
38      {
39        "_index": "shopping",
40        "_type": "shopping",
41        "_id": "AWJcYublQ1U_UGT8VEK",
42        "_score": 1,
43        "_source": {
44          "접수 번호": 392,
45          "주문 시간": "2017-11-08T10:06:40",
46          "수령 시간": "2017-11-09T08:43:40",
47          "예약 여부": "예약",
48          "배송 메모": "부재 중",
49          "고객 ip": "195.191.234.111",
50          "고객 성별": "남성",
51          "고객 나이": 21,
52          "물건 좌표": "35.93595097947761, 126.41956072432289",
53          "고객 주소_시도": "경상북도",
54          "구매 사이트": "위메프"
55        }
56      }
57    ]
58  }
59 }
```

## 과거에 작성한 API 이력 조회

The screenshot shows the Elasticsearch Dev Tools interface. On the left, the **Console** pane displays an Elasticsearch query:

```
1 GET /shopping/_search
2 [
3   "query": {
4     "match_all": {}
5   }
6 ]
```

A context menu is open over the query, with the option **Copy as cURL** highlighted. An arrow points from this menu to the **cURL 명령어로 복사** (Copy as cURL) label above.

The right side of the interface is the **Output Pane**, which shows the results of the executed query. The results are paginated, with page 1 of 1 shown. The first hit is a document with \_id AWJcYubilQ1U\_UGT8VEJ, containing fields like \_index, \_type, \_id, \_score, and \_source (which includes order details such as 접수 번호, 주문 시간, 수령 시간, 예약 여부, 배송 메모, 고객 ip, 고객 성별, 고객 나이, 물건 좌표, 고객 주소\_시도, 구매 사이트, 판매자 평점, 상품 분류, 상품 가격, 상품 개수, and 결제 카드).

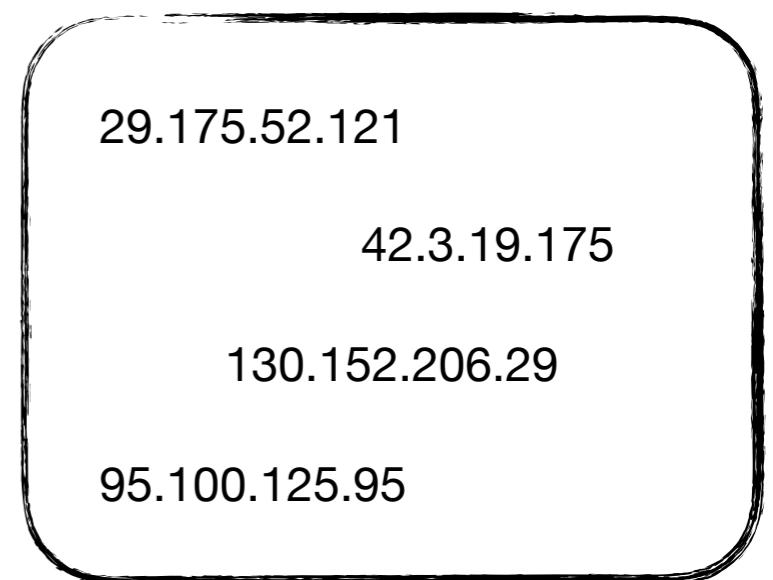
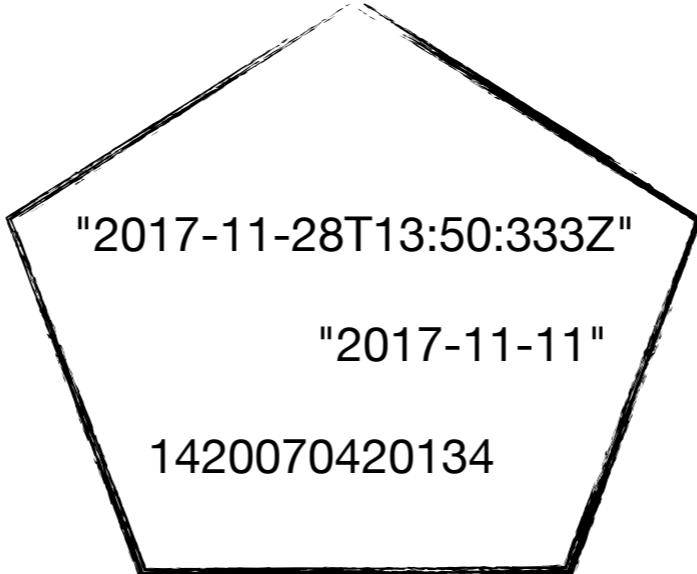
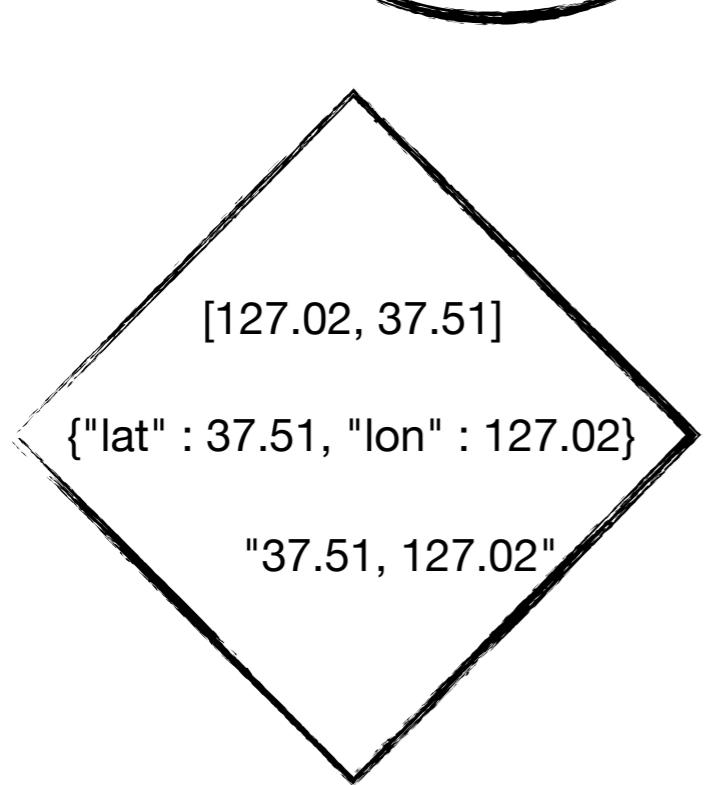
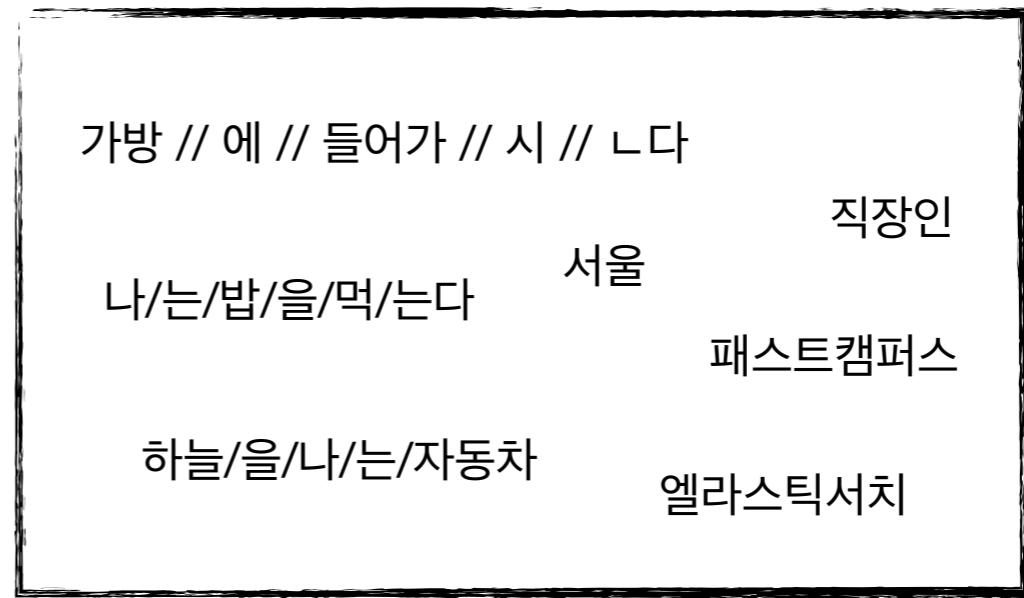
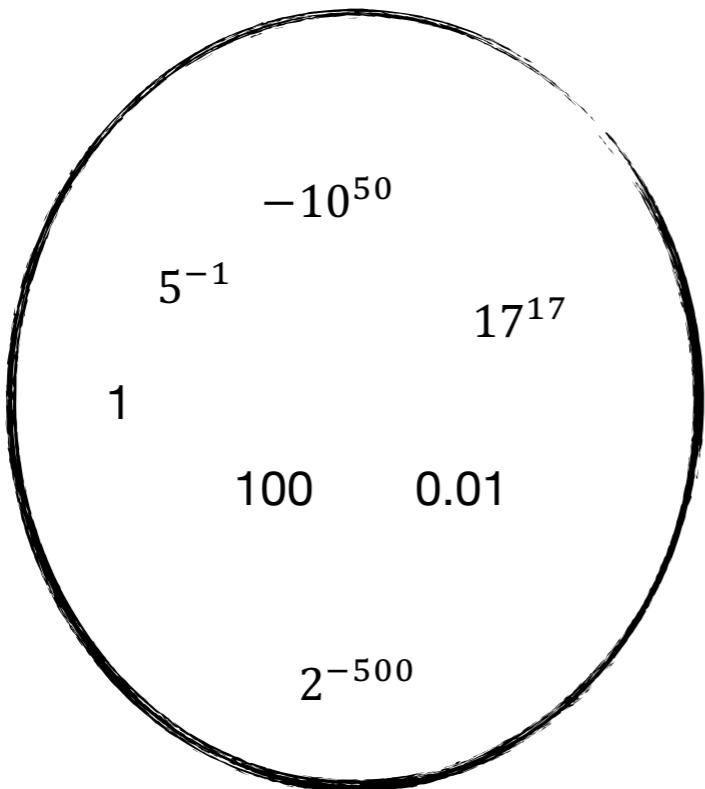
At the top right of the interface, there are tabs for **History**, **Settings**, and **Help**.

**Console : Elasticsearch API 작성**

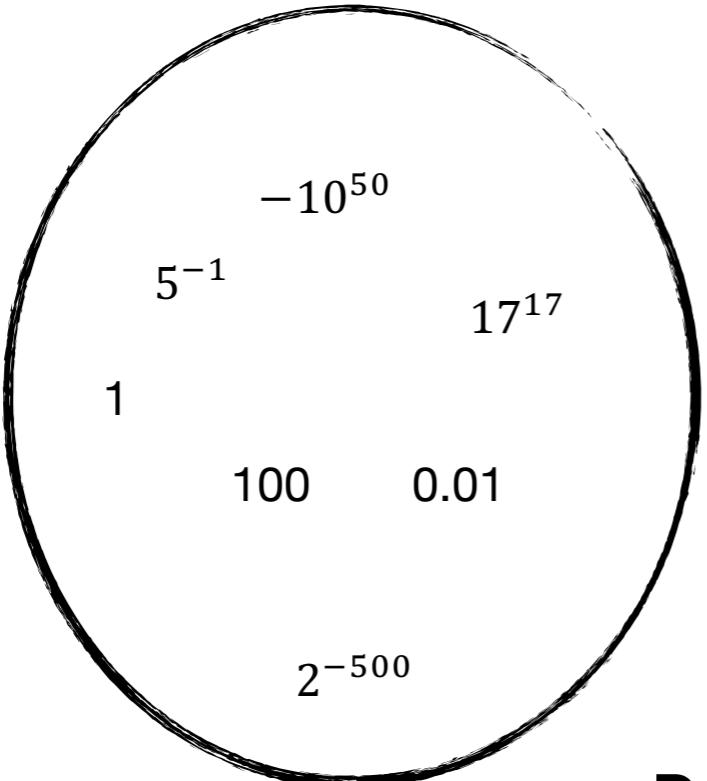
**cURL 명령어로 복사**

**Output Pane : 작성한 Elasticsearch API 결과 조회**

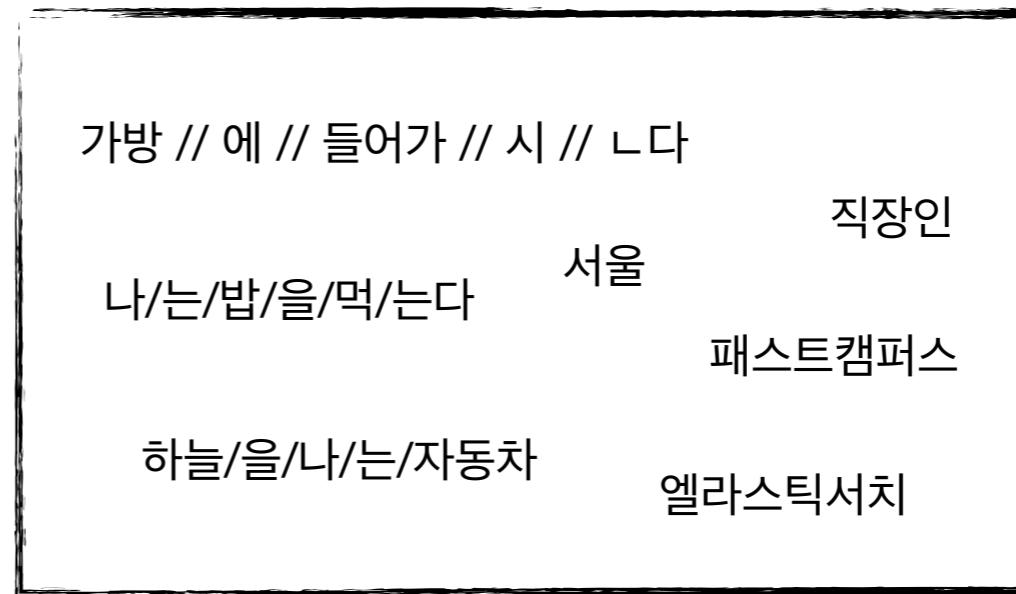
**Dashboard를 구축/운영 함에 있어  
알면 좋은 (최소한의) datatype을 살펴보자**



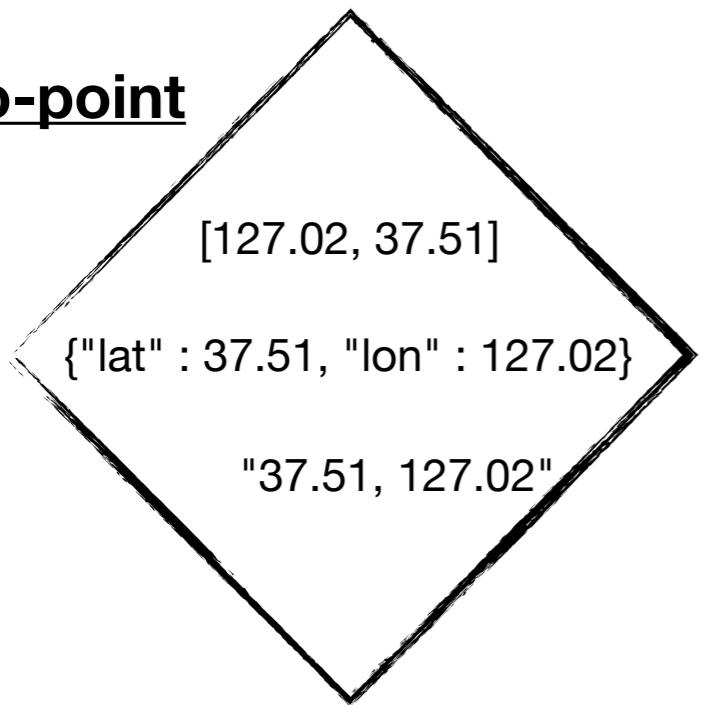
## Numeric



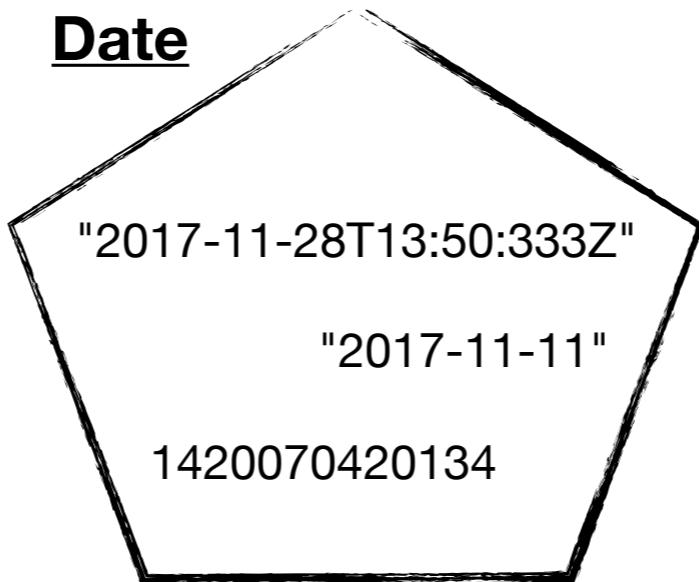
## String



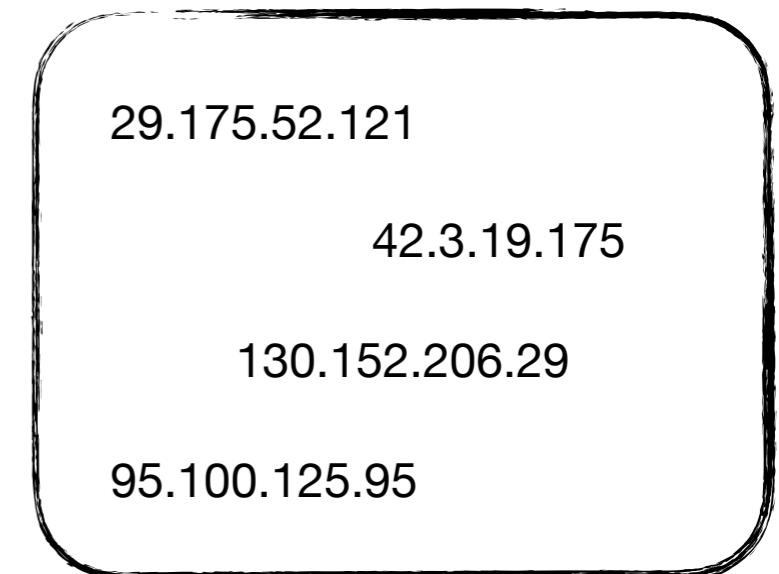
## Geo-point



## Date



## IP



이 때 주의할 Type은 **Numeric**과 **String**이다.

**Geo-point, Date, IP는 Format이 다양할 뿐 Type 자체는 1개다** 

## Numeric datatypes는 크게 정수형과 부동 소수점형으로 나뉜다



# Numeric datatypes

The following numeric types are supported:

정수

long	A signed 64-bit integer with a minimum value of $-2^{63}$ and a maximum value of $2^{63}-1$ .
integer	A signed 32-bit integer with a minimum value of $-2^{31}$ and a maximum value of $2^{31}-1$ .
short	A signed 16-bit integer with a minimum value of $-32,768$ and a maximum value of $32,767$ .
byte	A signed 8-bit integer with a minimum value of $-128$ and a maximum value of $127$ .

값의 범위

부동 소수점

double	A double-precision 64-bit IEEE 754 floating point.
float	A single-precision 32-bit IEEE 754 floating point.
half_float	A half-precision 16-bit IEEE 754 floating point.
scaled_float	A floating point that is backed by a long and a fixed scaling factor.

Precision 정도

# Numeric datatypes



The following numeric types are supported:

`long` A signed 64-bit integer with a minimum value of  $-2^{63}$  and a maximum value of  $2^{63}-1$ .

---

`integer` A signed 32-bit integer with a minimum value of  $-2^{31}$  and a maximum value of  $2^{31}-1$ .

---

`short` A signed 16-bit integer with a minimum value of  $-2^{15}$  and a maximum value of  $2^{15}-1$ .

---

## 어떤 Type을 사용해야 될까?

`byte` A signed 8-bit integer with a minimum value of  $-128$  and a maximum value of  $127$ .

---

`double` A double-precision 64-bit IEEE 754 floating point.

---

`float` A single-precision 32-bit IEEE 754 floating point.

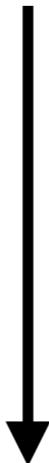
---

`half_float` A half-precision 16-bit IEEE 754 floating point.

---

`scaled_float` A floating point that is backed by a `long` and a fixed scaling factor.

가지고 있는 Numeric Data의 성격을 잘 모른다면,  
최적화에 집중하기보다는 데이터를 저장할 수 있는 범위가 가장 큰 type을 사용하자



- 정수형 : **Long**
- 부동소수점 : **Double**

## 가지고 있는 Numeric Data의 성격을 잘 안다면, 최적화에 집중하자



### 정수형

- 검색과 색인 성능 향상을 위해 데이터를 담을 수 있는 Smallest Type 고르자
- 다만 실제 저장된 값의 크기에 따라 용량이 정해지므로 어떤 Type을 고르던 Storage에는 영향은 없다
- Data Type 범위에 벗어나는 값은 저장할 수 없다 (p15)

### 부동소수점

- 데이터 왜곡을 허용할 수 있는 범위 내의 Smallest Type 고르자
- 어떤 Type을 사용하는지에 따라서 Disk Space에 많은 영향을 준다
- 데이터를 표현하기 부족한 Precision을 선택하면 예기치 않은 일이 생길 수 있다 (p16)

정수 : mapping에서 설정한 정수형 data type 범위 밖의 값을 넣을 경우 

PUT my\_index

```
{
  "mappings": {
    "my_type": {
      "properties": {
        "test": {
          "type": "byte"
        }
      }
    }
  }
}
```

POST my\_index/my\_type

```
{
  "test" : 129
}
```

```
{
  "error": {
    "root_cause": [
      {
        "type": "mapper_parsing_exception",
        "reason": "failed to parse [test]"
      }
    ],
    "type": "mapper_parsing_exception",
    "reason": "failed to parse [test]",
    "caused_by": {
      "type": "illegal_argument_exception",
      "reason": "Value [129] is out of range for a byte"
    }
  },
  "status": 400
}
```



## 부동소수점 : mapping에서 설정한 정수형 data type 범위 밖의 값을 넣을 경우

### 1) Field 생성

- test-double : double
- test-half-float : half-float

### 2) 데이터 입력

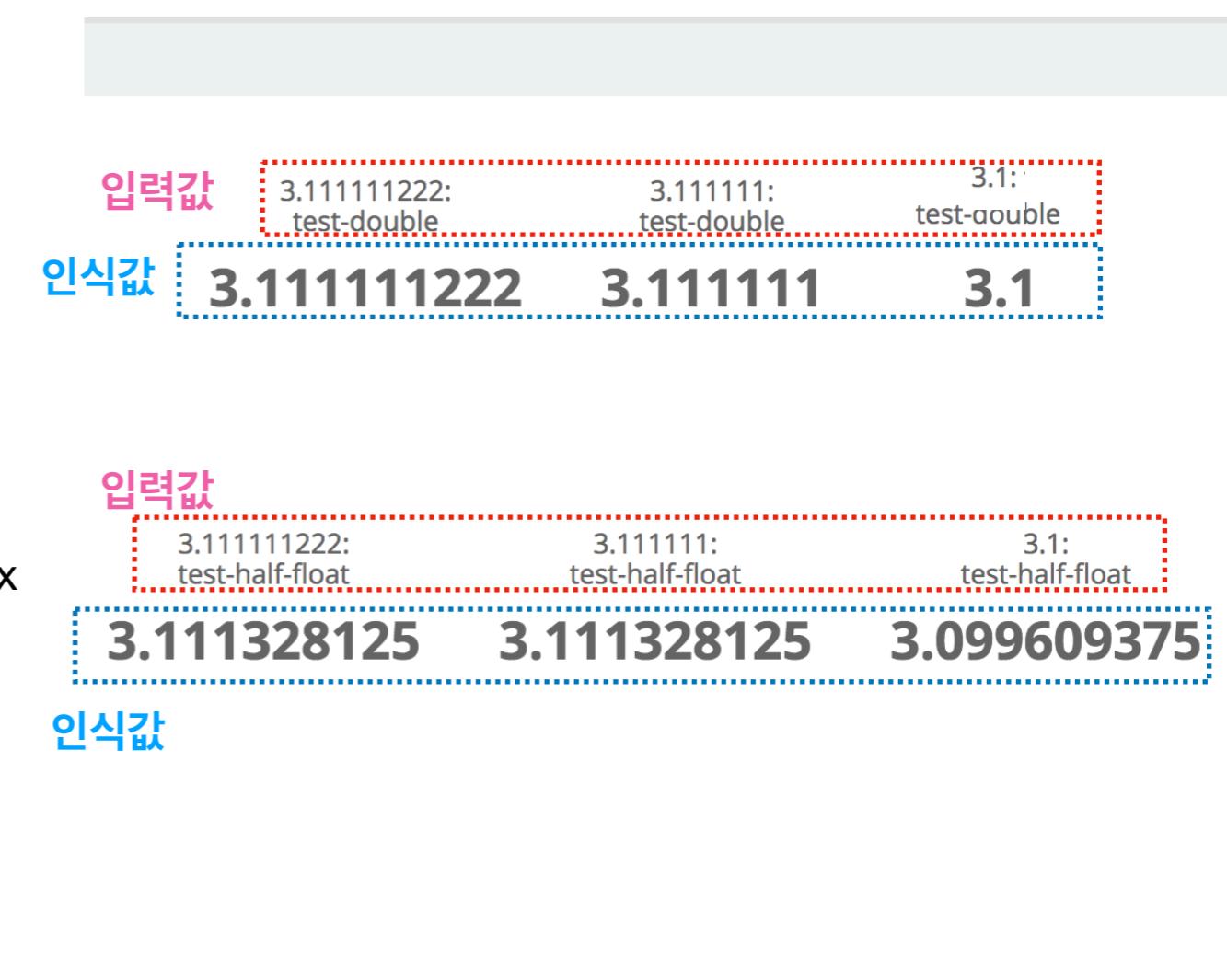
- test-double : 3.1, 3.111111, 3.11111122222
- test-half-float : 3.1, 3.111111, 3.11111122222

### 3) 데이터 검색

- test-double : 3.111111 ~ 3.1111112 값 검색 => 존재
- test-half-float : 3.111111 ~ 3.1111112 값 검색 => 존재 x

### 4) 데이터 확인

- test-double : 표현할 수 있는 Precision 내 존재
- test-half-float : 표현할 수 있는 Precision 내 존재 x



**Elasticsearch가 검색엔진인 만큼,  
String Field 선택은 매우 중요하다!**

**Keyword** : 입력 String Field의 값을 **하나의 단위**로 보고 싶은 경우 

**Text** : 입력 String Field를 **더 작은 단위**로 분석하고 싶은 경우 

입력 데이터

1) **Keyword**로 설정할 경우

2) **Text**로 설정 (분석기에 따라 상이)

가방에 들어가신다

가방에 들어가신다

가방 // 에 // 들어가 // 시 // 냈다

나는 밥을 먹는다

나는 밥을 먹는다

나 // 는 // 밥 // 을 // 먹 // 는다

패스트캠퍼스 엘라스틱서치

패스트캠퍼스 엘라스틱서치

패스트캠퍼스 // 엘라스틱서치



## "패스트캠퍼스 엘라스틱서치" => text field와 keyword field 비교

### 1) Field 생성

- test-text : text
- test-keyword : keyword

### 2) 데이터 입력

- test-text : "패스트캠퍼스 엘라스틱서치"
- test-keyword : "패스트캠퍼스 엘라스틱서치"

### 3) 데이터 조회

- test-text : "패스트캠퍼스" 조회
- test-keyword : "패스트컴퍼스" 조회

### 4) 데이터 확인

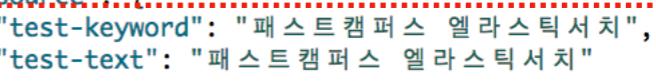
- test-text : 검색 o
- test-keyword : 검색 x

### test-text (검색됨)

```

"hits": {
  "total": 1,
  "max_score": 0.25811607,
  "hits": [
    {
      "_index": "my_index",
      "_type": "my_type",
      "_id": "AWE8BViSPloSIAlpN8ut",
      "_score": 0.25811607,
      "source": {
        "test-keyword": "패스트캠퍼스 엘라스틱서치",
        "test-text": "패스트캠퍼스 엘라스틱서치"
      }
    }
  ]
}

```

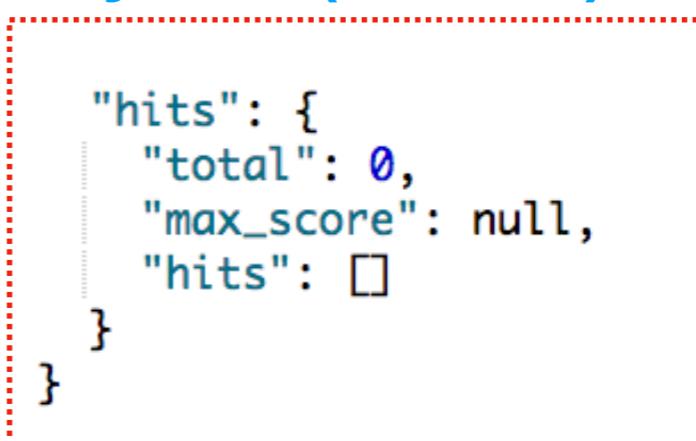



### test-keyword (검색 안됨)

```

"hits": {
  "total": 0,
  "max_score": null,
  "hits": []
}

```




~~API 학습 후에~~

# 조금 특별한 Data Type도 살펴보자 ( = Complex datatypes )

데이터를 계층적으로 저장할 수 없을까? 

POST **shopping/shopping**

```
{  
    "고객주소_시도": "서울특별시",  
    "상품": {  
        "가격": 27000,  
        "분류": "팬츠",  
        "개수": 7  
    }  
}
```

실제 색인 결과

```
{  
    "고객주소_시도": "서울특별시",  
    "상품.가격": 27000,  
    "상품.분류": "팬츠",  
    "상품.개수": 7  
}
```



"상품"이라는 inner object를 가지고 있다

```
PUT object
{
  "mappings": {
    "object": {
      "properties": {
        "고객주소_시도": {
          "type": "keyword"
        },
        "상품": {
          "properties": {
            "가격": { "type": "integer" },
            "분류": { "type": "keyword" },
            "개수": { "type": "integer" },
            }
          }
        }
      }
    }
}
```

데이터를 배열 형태로 저장할 수 없을까? 

POST **array/array**

```
{  
    "결제카드" : ["씨티", "국민"],  
    "고객성별" : "여성",  
    "상품" : [  
        {  
            "구매사이트" : "쿠팡",  
            "분류" : "셔츠"  
        },  
        {  
            "구매사이트" : "11번가",  
            "분류" : "팬츠"  
        }  
    ]  
}
```



실제 색인 결과

```
{  
    "고객성별" : "여성",  
    "결제카드" : ["씨티", "국민"],  
    "상품.구매사이트" : [ "쿠팡", "11번가" ],  
    "상품.분류" : [ "셔츠", "팬츠" ]  
}
```



```
PUT array
{
  "mappings": {
    "array": {
      "properties": {
        "결제카드": {
          "type": "keyword"
        },
        "고객성별": {
          "type": "keyword"
        },
        "상품": {
          "properties": {
            "구매사이트": { "type": "keyword" },
            "분류": { "type": "keyword" }
          }
        }
      }
    }
  }
}
```

데이터를 (앞과 살짝 다른) 배열 형태로 저장할 수 없을까 ? 

```
PUT nested
{
  "mappings": {
    "nested": {
      "properties": {
        "결제카드": {
          "type": "keyword"
        },
        "고객성별": {
          "type": "keyword"
        },
        "상품": {
          "type": "nested",
          "properties": {
            "구매사이트": { "type": "keyword" },
            "분류": { "type": "keyword" }
          }
        }
      }
    }
  }
}
```

POST **nested/nested**

```
{  
    "결제카드" : ["씨티", "국민"],  
    "고객성별" : "여성",  
    "상품" : [  
        {  
            "구매사이트" : "쿠팡",  
            "분류" : "셔츠"  
        },  
        {  
            "구매사이트" : "11번가",  
            "분류" : "팬츠"  
        }  
    ]  
}
```

### Nested datatype의 중요한 점 (!= array datatype)

object들이 field 별로 flatten되는 array datatype과 달리,  
상호 독립적으로 색인/검색될 수 있다!

**실제 예시를 통해 비교해보자**

```
PUT nested
{
  "mappings": {
    "nested": {
      "properties": {
        "결제카드": {
          "type": "keyword"
        },
        "고객성별": {
          "type": "keyword"
        },
        "상품": {
          "type": "nested",
          "properties": {
            "구매사이트": { "type": "keyword" },
            "분류": { "type": "keyword" }
          }
        }
      }
    }
  }
}
```

```
PUT array
{
  "mappings": {
    "array": {
      "properties": {
        "결제카드": {
          "type": "keyword"
        },
        "고객성별": {
          "type": "keyword"
        },
        "상품": {
          "properties": {
            "구매사이트": { "type": "keyword" },
            "분류": { "type": "keyword" }
          }
        }
      }
    }
  }
}
```

## POST nested/nested

```
{  
    "결제카드" : ["씨티", "국민"],  
    "고객성별" : "여성",  
    "상품" : [  
        {  
            "구매사이트" : "쿠팡",  
            "분류" : "셔츠"  
        },  
        {  
            "구매사이트" : "11번가",  
            "분류" : "팬츠"  
        }  
    ]  
}
```

## POST array/array

```
{  
    "결제카드" : ["씨티", "국민"],  
    "고객성별" : "여성",  
    "상품" : [  
        {  
            "구매사이트" : "쿠팡",  
            "분류" : "셔츠"  
        },  
        {  
            "구매사이트" : "11번가",  
            "분류" : "팬츠"  
        }  
    ]  
}
```

이 때 아래와 같은 검색을 한다고 했을 때 어떤 결과를 기대하는가?

상품.구매사이트 = 11번가 **그리고** 상품.분류 = 셔츠

3.1 array datatype 

```
GET array/_search
```

```
{  
  "query": {  
    "bool": {  
      "must": [  
        {  
          "match": {  
            "상품.구매사이트": "11번가"  
          }  
        },  
        {  
          "match": {  
            "상품.분류": "셔츠"  
          }  
        }  
      ]  
    }  
  }  
}
```

검색된다 !!

```
{  
  "took": 0,  
  "timed_out": false,  
  "_shards": {  
    "total": 5,  
    "successful": 5,  
    "skipped": 0,  
    "failed": 0  
  },  
  "hits": {  
    "total": 1,  
    "max_score": 0.5753642,  
    "hits": [  
      {  
        "_index": "array11",  
        "_type": "array11",  
        "_id": "AWLNYWnhzMQVnr-9MyPR",  
        "_score": 0.5753642,  
        "_source": {  
          "결제 카드": [  
            "씨티",  
            "국민"  
          ],  
          "고객 성별": "여성",  
          "상품": [  
            {  
              "구매 사이트": "쿠팡",  
              "분류": "셔츠"  
            },  
            {  
              "구매 사이트": "11번가",  
              "분류": "팬츠"  
            }  
          ]  
        }  
      }  
    ]  
  }  
}
```

## 3.1 array datatype

1) 다시 말해서, 원래 데이터는 아래와 같은 두 object를 가진 array 형태였다.

"상품.구매사이트" : "쿠팡",  
"상품.분류" : "셔츠"

"상품.구매사이트" : "11번가",  
"상품.분류" : "팬츠"

2) Elasticsearch는 위의 데이터가 아래의 조건을 만족한다고 판단한 것이다.

상품.구매사이트 = 11번가 **그리고** 상품.분류 = 셔츠

즉, **Association**을 무시해버리고 단순 value의 존재 유무만 고려한 것이다.

```
GET nested/_search
```

```
{  
  "query": {  
    "nested": {  
      "path": "상품",  
      "query": {  
        "bool": {  
          "must": [  
            {  
              "match": {  
                "상품.구매사이트": "11번가"  
              }  
            },  
            {  
              "match": {  
                "상품.분류": "셔츠"  
              }  
            }  
          ]  
        }  
      }  
    }  
  }  
}
```

검색이 안된다 !!



```
{  
  "took": 0,  
  "timed_out": false,  
  "_shards": {  
    "total": 5,  
    "successful": 5,  
    "skipped": 0,  
    "failed": 0  
  },  
  "hits": {  
    "total": 0,  
    "max_score": null,  
    "hits": []  
  }  
}
```

즉, **Association**을 고려해서 **object** 단위에서 조건을 만족하는 걸 판별한 것이다.

```
GET nested/_search
{
  "query": {
    "nested": {
      "path": "상품",
      "query": {
        "bool": {
          "must": [
            {
              "match": {
                "상품.구매사이트": "11번가"
              }
            },
            {
              "match": {
                "상품.분류": "팬츠"
              }
            }
          ]
        }
      }
    }
  }
}
```

검색이 된다 !!



```
{
  "took": 0,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "skipped": 0,
    "failed": 0
  },
  "hits": {
    "total": 1,
    "max_score": 1.3862944,
    "hits": [
      {
        "_index": "nested11",
        "_type": "nested11",
        "_id": "AWLNYPcTzMQVnr-9MyPQ",
        "_score": 1.3862944,
        "_source": {
          "결제 카드": [
            "씨티",
            "국민"
          ],
          "고객 성별": "여성",
          "상품": [
            {
              "구매사이트": "쿠팡",
              "분류": "셔츠"
            },
            {
              "구매사이트": "11번가",
              "분류": "팬츠"
            }
          ]
        }
      }
    ]
  }
}
```

즉, **Association**을 고려해서 **object** 단위에서 조건을 만족하는 걸 판별한 것이다.

**Array datatype과 Nested datatype은  
위 예시를 참고해서 목적에 맞게 사용하자**

# **Elastic Stack을 직접 설치/운영해보자**

# 어떤 방법으로 할까?

- Set up Elasticsearch

- Installing Elasticsearch

- Install Elasticsearch with [.zip or .tar.gz](#)

- Install Elasticsearch with [.zip on Windows](#)

**설치에 큰 시간 뺏기지 않고 누구나 같은 환경에서 작업할 수 있도록 Docker로 선정!** 

- Install Elasticsearch with RPM

- Install Elasticsearch with Windows MSI Insta

- Install Elasticsearch with Docker

- Administration, Monitoring, and Deployment

- + Monitoring

- Production Deployment

- Hardware

- Java Virtual Machine

- Transport Client Versus Node Client

- Configuration Management

- Important Configuration Changes

- Don't Touch These Settings!

- Heap: Sizing and Swapping

- File Descriptors and MMap

- Revisit This List Before Production

각자의 개발환경이 다르기에 실제 production에서도 만능 설정은 없기에 (없다고 믿으며),

실제 업무 현장에서 사용시에는 담당자들과 소통 후 환경을 설정하는 걸 권장한다.

설치 후 운영 중에도 끝없는 테스트가 수반되어야 한다.

# 우선 AWS EC2 Instance에 접속하자



Mac OS : Terminal  
Windows : Putty

**Elastic Stack**을 설치해보자  

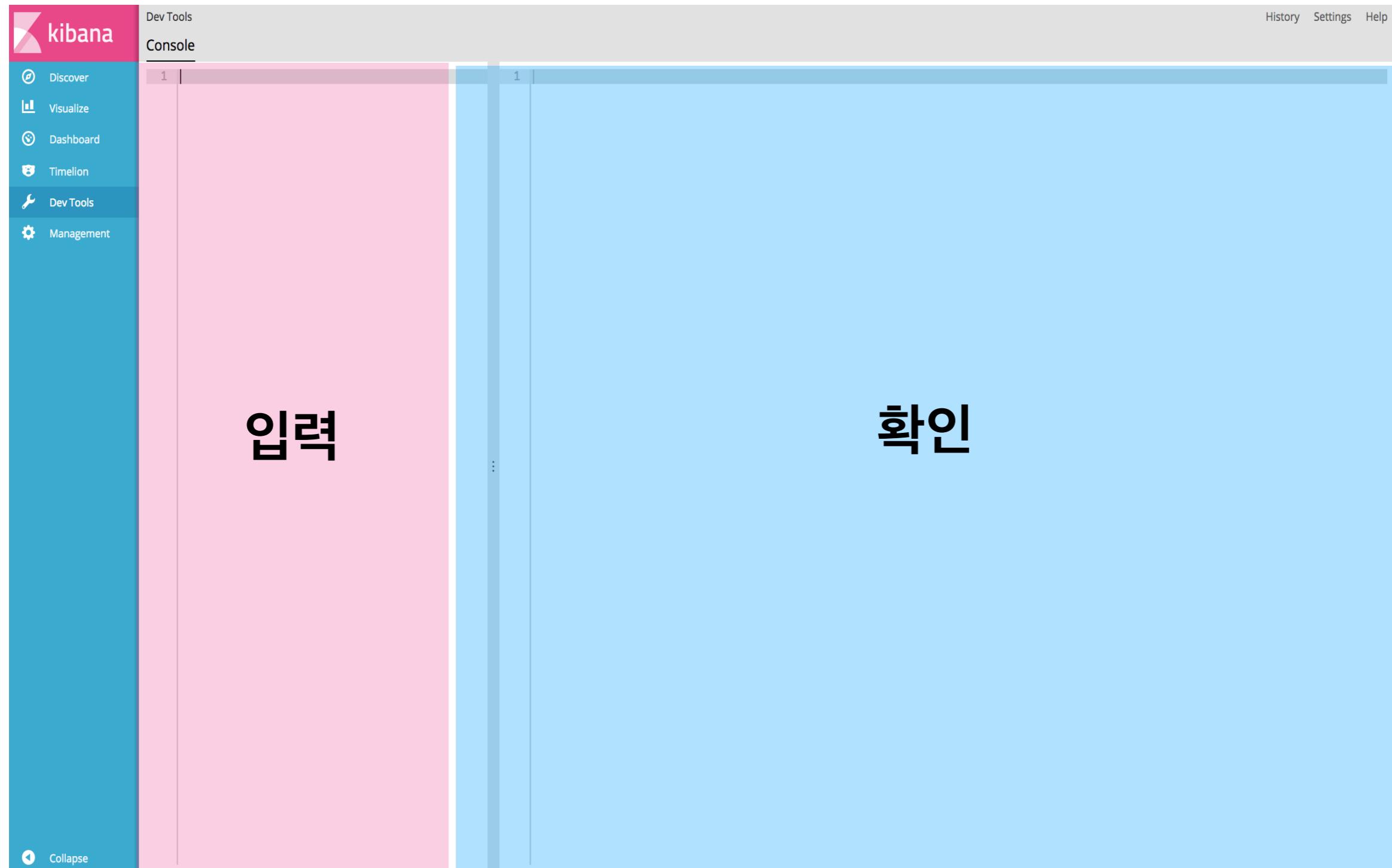
# 다양한 Elasticsearch API 중에서

## Indices, Document, Search API를 알아보자

**Indices API**로 무얼 할 수 있을까?



## Dev Tools 페이지로 이동해서 하나씩 입력하고 결과를 확인하자



## Index 생성

문법

PUT {Index 이름}

예시

PUT week4\_higee

## Index 삭제

문법

```
DELETE {Index 이름}
```

예시

```
DELETE week4_higee
```

Mapping API를 배우기 전에

## Mapping 설정은 꼭 필요한가?

Index의 Field들이 어떤 Type으로 저장되는지가 중요한가?

yes      no

필요하다

필요 없다

- Mapping 설정을 안해도 error가 발생하지는 않는다
- 단, 사용자가 원하는 Data Type으로 데이터가 저장된다는 보장이 없다. 예) "2017-01-01 13:00:00"
- 그러므로 (데이터 색인 전에) Mapping을 설정하는 걸 권장한다

## 그렇다면 Mapping은 어느 시점에 어떻게 설정하는가?

- Mapping을 통해 Data Type을 정의하려는 Field에 데이터가 색인되기 전까지는 아무 때나 가능하다
- Mapping을 설정 하기 전에 Data가 색인되면 Elasticsearch가 적당한 Data Type을 부여한다
  - 단, 한 번 설정된 Data Type은 변경이 불가능하다
  - 단, 데이터가 색인된 후에도 새로운 Field에 대한 Mapping은 추가할 수 있다
- 그러므로 일반적으로는 Index 생성하는 시점에 같이 설정하는 걸 권장한다

# Index 생성 후 Mapping 추가

문법

예시

```
PUT {Index 이름}
```

```
PUT week4_higee
```

```
PUT {Index 이름}/_mapping/{Type 이름}
{
  "properties": {
    "{Field 이름}" : {
      "type" : "{Field Type}"
    }
  }
}
```

```
PUT week4_higee/_mapping/week4_higee
{
  "properties": {
    "price" : {
      "type" : "integer"
    }
  }
}
```

# Index 생성하면서 Mapping 추가

문법

예시

```
DELETE {Index 이름}
```

```
DELETE week4_higee
```

(Index 생성하면서 Mapping 추가하는 실습을 위해 Index 삭제)

```
PUT {Index 이름}
```

```
{  
  "mappings": {  
    "{Type 이름)": {  
      "properties": {  
        "{Field1 이름)": {  
          "type": "{Field1 Type}"  
        },  
        "{Field2 이름)": {  
          "type": "{Field2 Type}"  
        }  
      }  
    }  
  }  
}
```

```
PUT week4_higee
```

```
{  
  "mappings": {  
    "week4_higee": {  
      "properties": {  
        "price": {  
          "type": "integer"  
        },  
        "time": {  
          "type": "date"  
        }  
      }  
    }  
  }  
}
```

# 기존 Mapping에 새로운 Field Mapping 추가하기

문법

```
PUT {Index 이름}/_mapping/{Type 이름}
{
  "properties": {
    "{Field 이름}" : {
      "type" : "{Field Type}"
    }
  }
}
```

예시

```
PUT week4_higee/_mapping/week4_higee
{
  "properties": {
    "age" : {
      "type" : "integer"
    }
  }
}
```

# Template 활용해서 자동으로 Mapping 추가

문법

예시

```
PUT _template/{Template 이름}
{
  "template": "{Index Pattern}",
  "mappings": {
    "{Type 이름)": {
      "properties": {
        "Field1 이름": {
          "type": "Field1 Type"
        },
        "Field2 이름": {
          "type": "Field2 Type"
        }
      }
    }
  }
}
```

```
PUT _template/template_higee
{
  "template": "higee-log-*",
  "mappings": {
    "template_higee": {
      "properties": {
        "price": {
          "type": "integer"
        },
        "time": {
          "type": "date"
        }
      }
    }
  }
}
```

**Template 생성 ≠ Index 생성**

**Template 생성 ≠ Mapping 생성**



Template에서 정의한 Index Pattern에 해당하는 Index가 생성될 때,  
Template을 활용해서 사전 정의한 Mapping이 적용된다

그렇다면 Template은 언제 유용할까?

비슷한 이름의 Index가 정기적으로 생성되는 Log Data 등

(higee-log-2018.01.01 higee-log-2018.01.02 higee-log-2018.01.03 ...)

Template을 사용하지 않으면

- 1) 자동으로 생성되는 Mapping을 사용하거나
- 2) 모든 Index마다 직접 Mapping을 추가해야 한다

## Mapping 확인

문법

예시

```
GET {Index 이름}/_mapping
```

```
GET week4_higee/_mapping
```

# Template Mapping 확인

문법

예시

PUT {Index 이름}

PUT **higee-log-2018.01.01**

GET {Index 이름}/\_mapping

GET **higee-log-2018.01.01/\_mapping**

## Index API 예제 + @

1. 실습 서버의 shopping index의 mapping을 확인하고,
2. 그와 같은 mapping 설정으로 자기 서버에서 shopping index를 생성하자
3. 완료한 경우, sli.do (#B083)에 ip주소를 적어주세요 -> **데이터 전송 목적 (Week5)**
4. dashboard  와 visualization  import
5. 다음의 scripted field 생성 
  - A. 배송소요시간 : ("수령시간" - "주문시간")/1000/60/60
  - B. 주문시간\_요일 : "주문시간"을 기준으로 '월', '화', ... '일' 표시
  - C. 주문시간\_시간대 : "주문시간"의 시간대 추출
  - D. 주문시간\_요일\_sort : "주문시간"의 요일 추출
  - E. 연령대 : "고객나이" 기준으로 10대, 20대, 30대, 40대, 50대 이상으로 표시

# **Document API로 무얼 할 수 있을까?**

**Document API**로 무얼 할 수 있을까?

- **Document 생성** ©
- **Document 조회** ®
- **Document 수정** ℗
- **Document 삭제** ®
- **Document 복사**

# Document 추가 (지정 ID)

문법

예시

```
PUT {Index 이름}/{Type 이름}/{ID}
{
  "{Field 이름}" : {Value}
}
```

```
PUT week4_higee/week4_higee/1
{
  "price" : 10000,
  "age" : 17
}
```

```
PUT week4_higee/week4_higee/2
{
  "price" : 2000,
  "age" : 20
}
```

```
PUT week4_higee/week4_higee/3
{
  "price" : 1000,
  "age" : 25
}
```

```
PUT week4_higee/week4_higee/4
{
  "price" : 7000,
  "age" : 33
}
```

# Document 추가 (임의 ID)

문법

예시

```
POST {Index 이름}/{Type 이름}
{
    "{Field 이름}" : {Value}
}
```

```
POST week4_higee/week4_higee
{
    "price" : 5000,
    "age" : 19
}
```

## ID로 Document 조회

\* Query로 Document 조회하는 건 Search API

문법

예시

```
GET {Index 이름}/{Type 이름}/{ID}
```

```
GET week4_higee/week4_higee/1
```

## ID로 Document 삭제

문법

예시

```
DELETE {Index 이름}/{Type 이름}/{ID}
```

```
DELETE week4_higee/week4_higee/1
```

# Query로 Document 삭제

문법

예시

```
POST {Index 이름}/_delete_by_query
{
  "query": {
    "match": {
      "{Field 이름)": "{Value}"
    }
  }
}
```

```
POST week4_higee/_delete_by_query
{
  "query": {
    "match": {
      "age": 20
    }
  }
}
```

# ID로 Document 부분 수정

문법

```
POST {Index 이름}/{Type 이름}/{ID}/_update  
{  
  "doc": {  
    "{Field}" : {Value}  
  }  
}
```

예시

```
POST week4_higee/week4_higee/3/_update  
{  
  "doc": {  
    "age" : 50  
  }  
}
```

# ID로 Document 전체 수정

문법

```
PUT {Index 이름}/{Type 이름}/{ID}
{
  "{Field}" : {Value}
}
```

예시

```
PUT week4_higee/week4_higee/3
{
  "warning" : "해당 Document 전체 변경"
}
```

# ID로 Document 수정 (Upsert)

문법

```
POST {Index 이름}/{Type 이름}/{ID}/_update
{
  "doc" : {
    "{Field 이름}" : "{Value}"
  },
  "doc_as_upsert" : true
}
```

예시

기존 Field Value 수정

```
POST week4_higee/week4_higee/4/_update
{
  "doc" : {
    "price" : 50000
  },
  "doc_as_upsert" : true
}
```

신규 Document 생성

```
POST week4_higee/week4_higee/777/_update
{
  "doc" : {
    "price" : 50000
  },
  "doc_as_upsert" : true
}
```

# Query로 Document 수정

문법

예시

```
POST {Index 이름}/{Type 이름}/_update_by_query
{
  "script": {
    "source": "ctx._source[{Field 이름}] = Value"
  },
  "query": {
    "term": {
      "{Field 이름)": "Value"
    }
  }
}
```

```
POST week4_higee/week4_higee/_update_by_query
{
  "script": {
    "source": "ctx._source['age'] = 50"
  },
  "query": {
    "term": {
      "age": 33
    }
  }
}
```

```
POST week4_higee/week4_higee/_update_by_query
{
  "script": {
    "source": "ctx._source.age = 70"
  },
  "query": {
    "term": {
      "age": 50
    }
  }
}
```

# Index 내 모든 Document 복사



문법

```
POST _reindex
{
  "source": {
    "index": "{복사하려는 원본 Index 이름}"
  },
  "dest": {
    "index": "{복사본을 저장할 Index 이름}"
  }
}
```

예시

```
POST _reindex
{
  "source": {
    "index": "week4_higee"
  },
  "dest": {
    "index": "week4_higee_reindex"
  }
}
```

## Reindex 사용 시 주의할 점

- Reindex 하는 순간 Destination Index는 생성된다
- Reindex는 순전히 Documents만 복사된다
- 그 외 Index 설정은 복사가 되지 않으므로 Destination Index 설정을 끝낸 후에 Reindex 사용 권장한다
- 즉, 가장 중요한 설정 중 하나인 **Mapping은 Reindex 전에 꼭 하기를 권장**한다

# Index 내 일부 Document 복사

문법

예시

```
POST _reindex
{
  "source": {
    "index": "{복사하려는 Index 이름}",
    "type" : "{복사하려는 Type 이름}",
    "query": {
      "term": {
        "{Field 이름}": "{Value}"
      }
    }
  },
  "dest": {
    "index": "{복사본을 저장할 Index 이름}"
  }
}
```

```
POST _reindex
{
  "source": {
    "index": "week4_higee",
    "type" : "week4_higee",
    "query": {
      "term": {
        "age": 19
      }
    }
  },
  "dest": {
    "index": "week4_higee_reindex2"
  }
}
```

# Search API (특히 Query DSL)로 무얼 할 수 있을까?



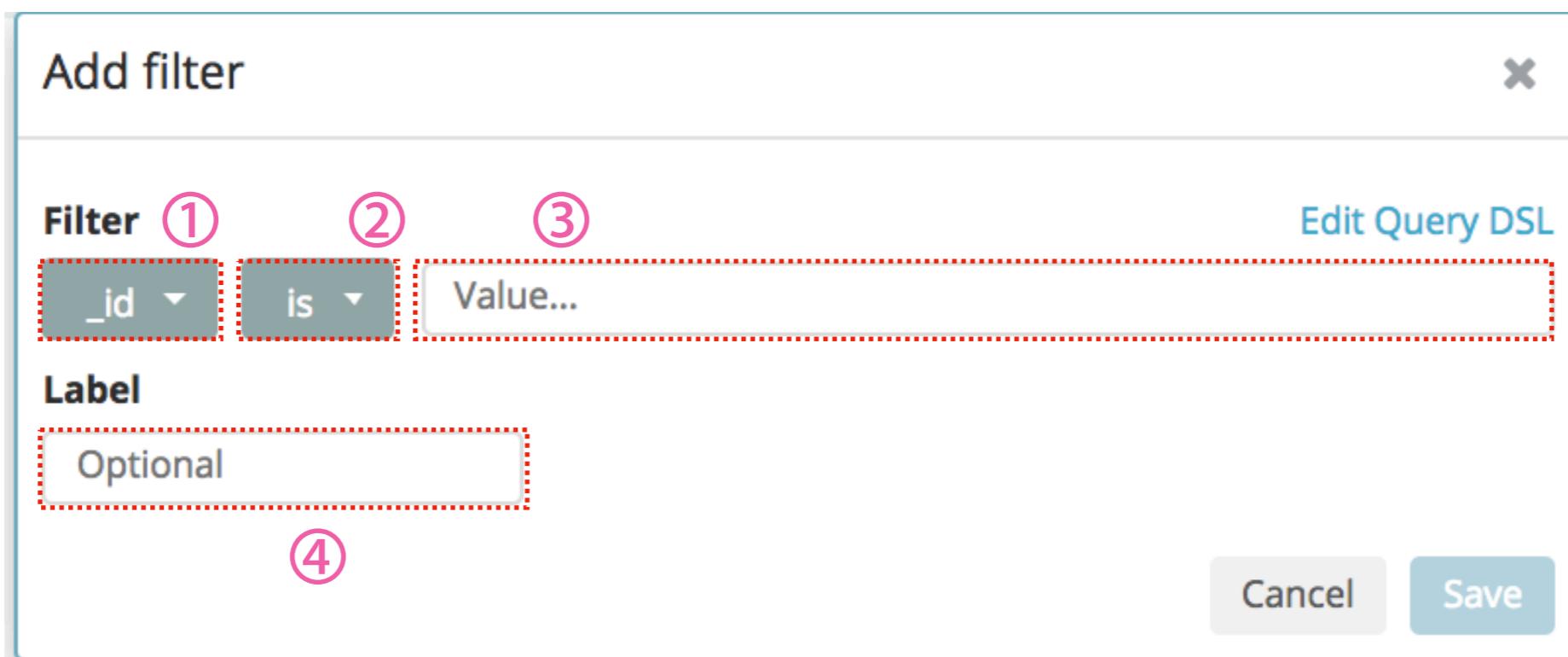
Request Body Search에서 사용되는 Domain Specific Language

Match All Query	Full Text Queries	Term Level Queries	Specialized Queries	Compound Queries
match-all	match query-string ⋮	exists fuzzy prefix range term terms wildcard ⋮	script ⋮	bool ⋮

간략히나마 뭘 위한건지는 알겠는데  
Dashboard를 구축/운영하는데 왜 필요하지?

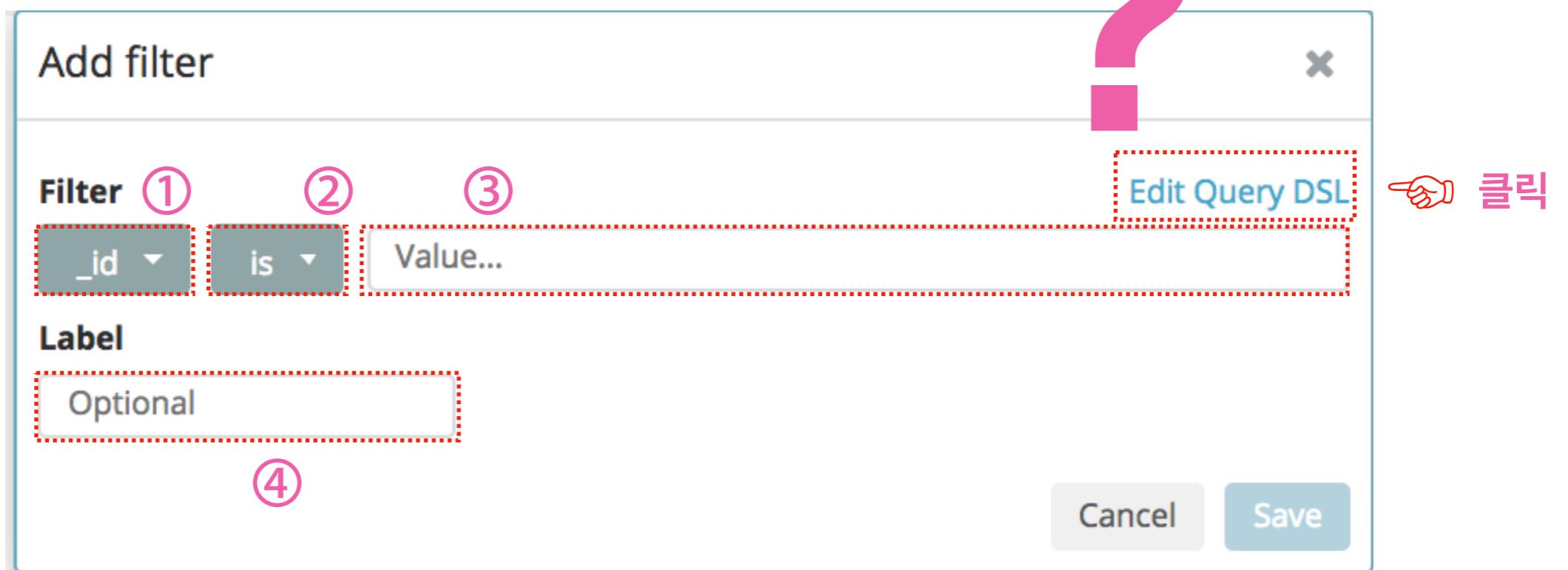
**Filter 기능 강화를 위해서**

## Filter를 다시 보자



- ① Filter 적용할 Field 선택
- ② 적용할 Operator 선택 (다음 페이지 참조)
- ③ Filter에 적용하려는 Value 입력
- ④ (여러 Filter 구분하기 위한) 이름 입력

## Filter를 다시 보자



- ① Filter 적용할 Field 선택
- ② 적용할 Operator 선택 (다음 페이지 참조)
- ③ Filter에 적용하려는 Value 입력
- ④ (여러 Filter 구분하기 위한) 이름 입력

## Edit Query DSL

Add filter ×

Filter

Search filter values

1 { }

이 부분에 **Query DSL**을 활용해서  
Filter를 생성할 수 있다.

Filters are built using the [Elasticsearch Query DSL](#).

Label

Optional

Cancel

Save

## 예를 들어, 아래와 같은 Query DSL을 작성하면

Add filter



### Filter

```
1 {  
2   "query": {  
3     "term": {  
4       "상품분류": "셔츠"  
5     }  
6   }  
7 }
```

Search filter values

☞ 1. 입력

Filters are built using the [Elasticsearch Query DSL](#).

### Label

셔츠

☞ 2. 입력

Cancel

Save

☞ 3. 선택

# 이런 결과가 나온다

Kibana 135 hits Search... (e.g. status:200 AND extension:PHP) New Save Open Share < ⌂ Last 90 days > Uses lucene query syntax Actions!

Discover Visualize Dashboard Timelion Dev Tools Management

Selected Fields Available Fields Popular

t 결제카드 t 고객주소\_시도 t 배송메모 # 배송소요시간 t 상품분류 ⓧ 수령시간 # 시간대 t 연령대 t 요일 ⓧ 주문시간 t \_id t \_index # \_score t \_type t 결제카드- t 결제카드풀네이 t 고객ip # 고객나이 t 고객성별 t 구매사이트 t 나이를알려주겠ㄷ ⓧ 물건좌표 # 배송소요시간-

셔츠 Add Order

## 적용완료

October 31st 2017, 03:38:08.352 - January 29th 2018, 03:38:08.352 — Auto

Count 주문시간 per day

Time \_source

데이터 확인

▶ 12월31일 17시59분 상품분류: 셔츠 접수번호: 759 주문시간: 12월31일 17시59분 수령시간: 01월04일 18시24분 예약여부: 일반 배송메모: 부재중 고객ip: 198.69.83.206 고객성별: 여성 고객나이: 26 물건좌표: 37.82120780752847, 128.65164454811415 고객주소\_시도: 서울특별시 구매사이트: 위메프 판매자평점: 1 상품가격: 28,000 상품개수: 7 결제카드: 우리 \_id: AV-iDM6XRJy4v-Hns1 aG \_type: shopping \_index: shopping \_score: - 배송이얼마나걸리나: 96 요일-: 7 성별카드가자: 여성-우리 배송소요일수: 나흘 이상 배송소요일수\_sort: 3 나이를알려주겠ㄷ: 20 age 시간대: 8 연령대: 20대 소비행태: 과소비 배송소요시간: 96 요일\_sort: 7 결제카드풀네이: 우리카드 요일: 일 성별-카드: 여성-우리 배송소요시간-: 96 결제카드-: 우리카드

▶ 12월30일 11시34분 상품분류: 셔츠 접수번호: 5413 주문시간: 12월30일 11시34분 수령시간: 01월01일 21시40분 예약여부: 일반 배송메모: 관리실에 맡김 고객ip: 80.220.204.239 고객성별: 남성 고객나이: 2 1 물건좌표: 35.048275031549871, 128.07056814953543 고객주소\_시도: 대전광역시 구매사이트: GS샵 판매자평점: 1 상품가격: 18,000 상품개수: 1 결제카드: 국민 \_id: AV-iDkN5RJy4v-Hns2i0 \_type: shopping \_index: shopping \_score: - 배송이얼마나걸리나: 58 요일-: 6 성별카드가자: 남성-국민 배송소요일수: 모레 배송소요일수\_sort: 2 나이를알려주겠ㄷ: 20 age 시간대: 2 연령대: 20대 소비행태: 저소비 배송소요시간: 58 요일\_sort: 6 결제카드풀네이: 국민카드 요일: 토 성별-카드: 남성-국민 배송소요시간-: 58 결제카드-: 국민카드

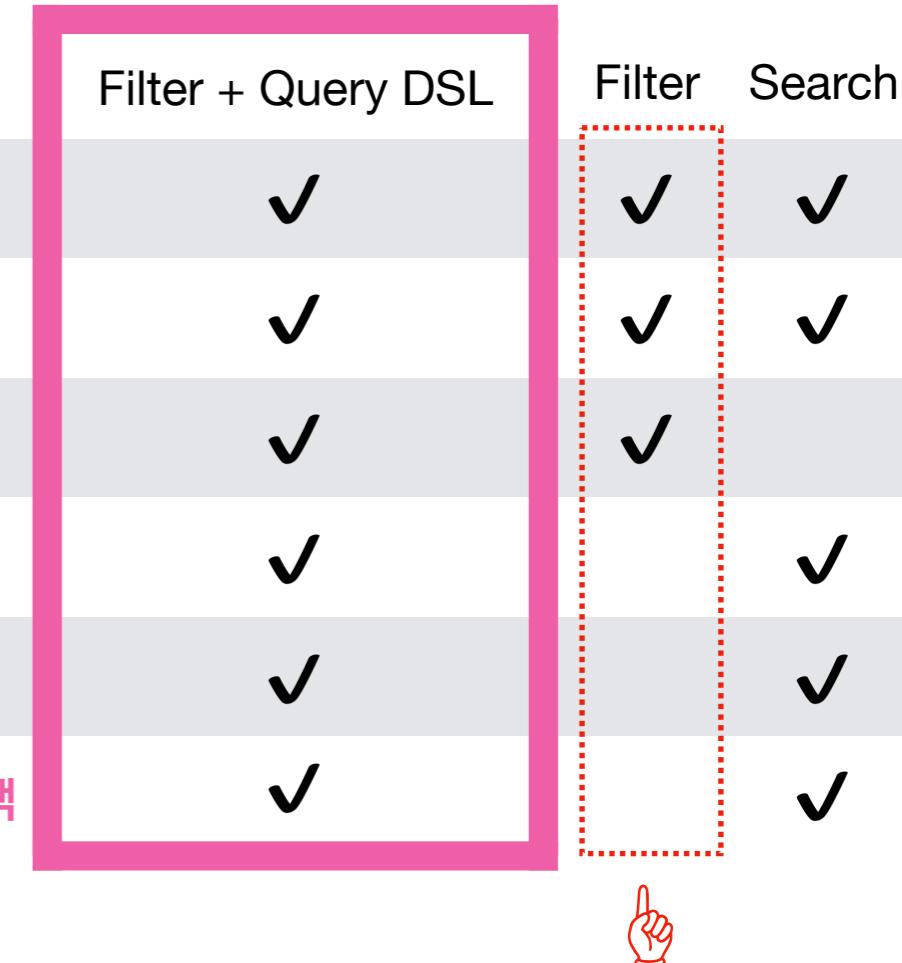
▶ 12월30일 07시07분 상품분류: 셔츠 접수번호: 3451 주문시간: 12월30일 07시07분 수령시간: 01월02일 14시37분 예약여부: 일반 배송메모: 부재중 고객ip: 167.190.90.160 고객성별: 여성 고객나이: 24 물건좌표: 37.346744741125214, 128.5509722697623 고객주소\_시도: 충청남도 구매사이트: 쿠팡 판매자평점: 1 상품가격: 20,000 상품개수: 1 결제카드: 우리 \_id: AV-iDaTSRJy4v-Hns2EK \_type: shopping \_index: shopping \_score: - 배송이얼마나걸리나: 79 요일-: 5 성별카드가자: 여성-우리 배송소요일수: 나흘 이상 배송소요일수\_sort: 3 나이를알려주겠ㄷ: 20 age 시간대: 22 연령대: 20대 소비행태: 저소비 배송소요시간: 79 요일\_sort: 5 결제카드풀네이: 우리카드 요일: 금 성별-카드: 여성-우리 배송소요시간-: 79 결제카드-: 우리카드

▶ 12월29일 19시11분 상품분류: 셔츠 접수번호: 8534 주문시간: 12월29일 19시11분 수령시간: 01월02일 12시59분 예약여부: 일반 배송메모: 관리실에 맡김 고객ip: 25.182.190.33 고객성별: 여성 고객나이: 17 물건좌표: 36.22963636912366, 127.40979543882128 고객주소\_시도: 전라남도 구매사이트: 쿠팡 판매자평점: 1 상품가격: 22,000 상품개수: 1 결제카드: 하나 \_id: AV-iD0VvRJy4v-Hn s3TL \_type: shopping \_index: shopping \_score: - 배송이얼마나걸리나: 89 요일-: 5 성별카드가자: 여성-하나 배송소요일수: 나흘 이상 배송소요일수\_sort: 3 나이를알려주겠ㄷ: 10 age 시간대: 10 연령대: 10대 소비행태: 저소비 배송소요시간: 89 요일\_sort: 5 결제카드풀네이: 하나카드 요일: 금 성별-카드: 여성-하나 배송소요시간-: 89 결제카드-: 하나카드

▶ 12월29일 11시48분 상품분류: 셔츠 접수번호: 13004 주문시간: 12월29일 11시48분 수령시간: 01월02일 07시44분 예약여부: 일반 배송메모: 주소 오류 고객ip: 92.198.64.230 고객성별: 여성 고객나이: 20 물건좌표: 37.297368471075586, 126.27321074940016 고객주소\_시도: 제주특별자치도 구매사이트: GS샵 판매자평점: 4 상품가격: 17,000 상품개수: 1 결제카드: 우리 \_id: AV-iE5-RJ y4v-Hns4Zc \_type: shopping \_index: shopping \_score: - 배송이얼마나걸리나: 91 요일-: 5 성별카드가자: 여성-우리 배송소요일수: 나흘 이상 배송소요일수\_sort: 3 나이를알려주겠ㄷ: 20 age 시간대: 2 연령대: 20대 소비행태: 저소비 배송소요시간: 91 요일\_sort: 5 결제카드풀네이: 우리카드 요일: 금 성별-카드: 여성-우리 배송소요시간-: 91 결제카드-: 우리카드

## 즉, Filter + Query DSL을 이용하면

		Filter + Query DSL	Filter	Search
"고객성별"이 여성인 Data		✓	✓	✓
"결제카드"가 우리 또는 국민인 Data		✓	✓	✓
"고객성별"이 남성이면서 "연령대"가 20대	SCRIPTED FIELD	✓	✓	
"구매사이트"가 쿠팡이거나 "상품개수"가 1~3인 Data	OR 연산	✓		✓
"결제카드"가 "우"로 시작하는 모든 Data	WILDCARD 검색	✓		✓
"구매사이트"가 22번가(오타 아니에요)와 유사한 Data	FUZZY / PROXIMITY 검색	✓		✓



- 지난 수업 때 얘기한 Filter는 Query DSL 제외
- 수업 진도상 표기일 뿐 실제로는 둘 다 Filter로 본다

# 모든 Documents 조회

문법

```
GET {Index 이름}/{Type 이름}/_search
{
  "query" : {
    "match_all" : {}
  }
}
```

예시

```
GET shopping/shopping/_search
{
  "query" : {
    "match_all" : {}
  }
}
```

```
{  
① "took": 0,          ②  
  "timed_out": false,  
  "_shards": {  
    "total": 5,  
    "successful": 5,  
    "skipped": 0,  
    "failed": 0  
  },                ③  
  "hits": {  
    "total": 20222,  
    "max_score": 1,  
    "hits": [  
      {  
        "_index": "shopping",  
        "_type": "shopping",  
        "_id": "AV-iDKZcRJy4v-Hns1Sk",  
        "_score": 1,  
        "_source": {  
          "접수번호": "277",  
          "주문시간": "2016-04-11T04:28:14",  
          "고객ip": "130.152.206.29",  
          "물건좌표": "36.56, 129.87",  
          "판매자평점": 3,  
          "상품분류": "스웨터",  
          "상품가격": 10000,  
        }  
      }  
    ]  
  }  
}
```



- ① Elasticsearch 검색 소요시간 (millisecond)
- ② 검색결과가 time out에 걸렸는지 표시
- ③ 몇 개의 shards가 검색되었는지 표시
- ④ 검색된 Documents의 개수
- ⑤ 실제 Documents 내용

# 검색어와 정확히 일치하는 value를 가진 Document 조회

문법

```
GET {Index 이름}/{Type 이름}/_search
{
  "query" : {
    "term" : {
      "{Field 이름}" : "{Value}"
    }
  }
}
```

예시

```
GET shopping/shopping/_search
{
  "query" : {
    "term" : {
      "상품분류" : "셔츠"
    }
  }
}
```

## 검색어 중 적어도 1개와 정확히 일치하는 Document 조회

문법

```
GET {Index 이름}/{Type 이름}/_search
{
  "query" : {
    "terms" : {
      "{Field 이름}" : [
        "{Value}", "{Value}"
      ]
    }
  }
}
```

예시

```
GET shopping/shopping/_search
{
  "query" : {
    "terms" : {
      "상품분류" : [
        "셔츠", "스웨터"
      ]
    }
  }
}
```

# 특정 접두어로 시작하는 Document 조회



문법

```
GET {Index 이름}/{Type 이름}/_search
{
  "query": {
    "prefix" : {
      "{Field 이름}" : "{Value}"
    }
  }
}
```

예시

```
GET shopping/shopping/_search
{
  "query": {
    "prefix" : {
      "고객주소_시도" : "경상"
    }
  }
}
```

# Wildcard Expression 만족하는 Documents 조회



문법

```
GET {Index 이름}/{Type 이름}/_search
{
  "query": {
    "wildcard" : {
      "{Field 이름}" : "{Value}"
    }
  }
}
```

예시

```
GET shopping/shopping/_search
{
  "query": {
    "wildcard" : {
      "고객주소_시도" : "경*도"
    }
  }
}
```

```
GET shopping/shopping/_search
{
  "query": {
    "wildcard" : {
      "고객주소_시도" : "경?도"
    }
  }
}
```

# 검색어와 유사한 Documents 조회

문법

```
GET {Index 이름}/{Type 이름}/_search
{
  "query": {
    "fuzzy" : {
      "{Field 이름}" : "{Value}"
    }
  }
}
```

예시

```
GET shopping/shopping/_search
{
  "query": {
    "fuzzy" : {
      "고객주소_시도" : {
        "value" : "경상북남"
      }
    }
  }
}
```

# 특정 Numeric Field가 임의의 범위 내에 있는 Documents 조회

문법

```
GET {Index 이름}/{Type 이름}/_search
{
  "query": {
    "range": {
      "{Field 이름)": {
        "gte": "{Value}",
        "lte": "{Value}",
      }
    }
  }
}
```

예시

```
GET shopping/shopping/_search
{
  "query": {
    "range": {
      "주문시간": {
        "gte": "2017-02-15"
      }
    }
  }
}
```

# non-null value가 존재하는 Documents 조회



문법

```
GET {Index 이름}/{Type 이름}/_search
{
  "query": {
    "exists" : {
      "field" : "{Field 이름}"
    }
  }
}
```

예시

```
GET shopping/shopping/_search
{
  "query": {
    "exists" : {
      "field" : "상품분류"
    }
  }
}
```

# 검색어와 부분적으로 일치하는 value를 가진 Document 조회



문법

```
GET {Index 이름}/{Type 이름}/_search
{
  "query": {
    "match": {
      "{Field 이름)": "{value}"
    }
  }
}
```

예시

```
GET shopping/_search
{
  "query": {
    "match": {
      "배송메모": "배송 못함"
    }
  }
}
```

```
GET shopping/_search
{
  "query": {
    "match": {
      "배송메모": "시간 못함"
    }
  }
}
```

# Lucene Query Syntax를 만족하는 Documents 조회



문법

```
GET {Index 이름}/{Type 이름}/_search
{
  "query" : {
    "query_string" : {
      "query" : "{LUCENE QUERY}"
    }
  }
}
```

예시

```
GET shopping/shopping/_search
{
  "query" : {
    "query_string" : {
      "query": "고객나이 : [10 TO 25]"
    }
  }
}
```



Query String Syntax

# Scripted Field가 특정 조건을 만족하는 Document 조회

The screenshot shows the Kibana Management interface with the title "Management / Kibana". Under "Index Patterns", the "shopping" pattern is selected. The page displays a list of fields, with "scripted fields (16)" highlighted. A red dashed box encloses the list of 16 Scripted fields, which are listed below:

name	lang	script	format	controls
시간대	painless	doc['주문시간'].date.hourOfDay		
배송소요시	painless	(doc['수령시간'].value-doc['주문시간'].value)/1000/60/60 간		
요일	painless	( doc['주문시간'].date.dayOfWeek == 1 ? '월' : (doc['주문시간'].date.dayOfWeek == 2 ? '화' : ((doc['주문시간'].date.dayOfWeek == 3 ? '수' : ((doc['주문시간'].date.dayOfWeek == 4 ? '목' : ((doc['주문시간'].date.dayOfWeek == 5 ? '금' : ((doc['주문시간'].date.dayOfWeek == 6 ? '토' : '일'))))))))) )		
요일_sort	painless	doc['주문시간'].date.dayOfWeek		
연령대	painless	if(doc['고객나이'].value < 20) { return "10대" } else if (doc['고객나이'].value < 30) { return "20대" } else if (doc['고객나이'].value < 40) { return "30대" } else if (doc['고객나이'].value < 50) { return "40대" } return "50대 이상"		
배송소요일	painless	if((doc['수령시간'].value-doc['주문시간'].value)/1000/60/60 < 24) { return "당일" } else if ((doc['수령시간'].value-doc['주문시간'].value)/1000/60/60 < 48) { return "다음날" } else if ((doc['수령시간'].value-doc['주문시간'].value)/1000/60/60 < 72) { return "모레" } return "나흘 이상"		
배송소요일	painless	if((doc['수령시간'].value-doc['주문시간'].value)/1000/60/60 < 24) { return 0 } if ((doc['수령시간'].value-doc['주문시간'].value)/1000/60/60 < 48) { return 1 } if ((doc['수령시간'].value-doc['주문시간'].value)/1000/60/60 < 72) { return 2 } return 3		
성별-카드	painless	doc['고객성별'].value + '-' + doc['결제카드'].value	String	
결제카드-	painless	doc['결제카드'].value + '카드'	String	
배송소요시	painless	(doc['수령시간'].value-doc['주문시간'].value)/1000/60/60 간-	Number	
소비행태	painless	if(doc['상품개수'].value < 3) { return "저소비" } else if (doc['고객나이'].value < 6) { return "평균" } return "과소비"	String	
요일-	painless	doc['주문시간'].date.dayOfWeek		
성별카드가	painless	doc['고객성별'].value + '-' + doc['결제카드'].value 자	String	
결제카드풀	painless	doc['결제카드'].value + "카드" 네이	String	



위에서 생성했던 Scripted Field를 직접 사용하지는 못한다

# Script Query가 특정 조건을 만족하는 Document 조회



문법

```
GET {Index 이름}/{Type 이름}/_search
{
  "query": {
    "script": {
      "script": {
        "source": "{Script Field 및 조건}",
        "lang": "painless"
      }
    }
  }
}
```

예시

```
GET shopping/_search
{
  "query": {
    "script": {
      "script": {
        "source": "doc['주문시간'].date.hourOfDay > 15",
        "lang": "painless"
      }
    }
  }
}
```

 조건

 Scripted Field 생성 Source

즉, **Scripted Field** 생성하기 위해 사용했던 코드를 직접 입력해야 한다

# 여러가지 Query를 복합적으로 사용할 수 있을까?

- A : 고객주소\_시도 = 서울특별시
- B : 구매사이트 = 11로 시작
- C : 고객나이 < 30
- D : 주문날짜 = 일요일

**Term Query**  
**Wildcard Query**  
**Range Query**  
**Script Query**



위의 Query를 아래와 같은 조건으로 검색 가능

- A AND B
- A AND NOT B
- A OR B
- A AND (B OR C)
- A AND (B OR C OR D 중 2개 이상 만족)

⋮

## Bool Query의 Occurrence Type을 알아보자

Bool Query Occurrence	Logical Statement
must	AND
must_not	NOT
should	OR

위의 비교가 정확히 일치하지는 않으니 참고로만 하자

# 기본 구조는 다음과 같다 (Term Query 예시)

```
GET {Index 이름}/{Type 이름}/_search
```

```
{  
  "query": {  
    "bool": {  
      "must": [  
        {  
          "term" : {  
            "고객주소_시도" : "서울특별시"  
          }  
        }  
      ],  
      "must_not": [  
        {  
          "term" : {  
            "상품분류" : "셔츠"  
          }  
        }  
      ],  
      "should": [  
        {  
          "term": {  
            "결제카드": "시티"  
          }  
        }  
      ],  
      "minimum_should_match": 1  
    }  
  }  
}
```

👉 반드시 만족해야 한다

👉 반드시 만족하면 안된다

👉 {minimum\_should\_match}개 이상 만족해야 한다

👉 should clause 내의 query가 n개 이상 참이어야 한다

# 1. A AND B를 구해보자

A : 고객주소\_시도 = 서울특별시  
B : 구매사이트 = 11로 시작

```
GET shopping/_search
{
  "query": {
    "bool": {
      "must": [
        { "term": { "고객주소_시도": "서울특별시" } },
        { "wildcard": { "구매사이트": "11*" } }
      ]
    }
  }
}
```

## 2. A AND NOT B를 구해보자

A : 고객주소\_시도 = 서울특별시  
B : 구매사이트 = 11로 시작

```
GET shopping/_search
{
  "query": {
    "bool": {
      "must": [
        { "term": { "고객주소_시도": "서울특별시" } }
      ],
      "must_not": [
        { "wildcard": { "구매사이트" : "11*" } }
      ]
    }
  }
}
```

### 3. A OR B를 구해보자

A : 고객주소\_시도 = 서울특별시  
B : 구매사이트 = 11로 시작

```
GET shopping/_search
{
  "query": {
    "bool": {
      "should": [
        { "term": { "고객주소_시도": "서울특별시" }},
        { "wildcard": { "구매사이트" : "11*"}}
      ],
      "minimum_should_match": 1
    }
  }
}
```

## 4. A AND (B OR C)를 구해보자

A : 고객주소\_시도 = 서울특별시  
B : 구매사이트 = 11로 시작  
C : 고객나이 < 30

```
GET shopping/_search
{
  "query": {
    "bool": {
      "must": [
        { "term": { "고객주소_시도": "서울특별시" }}
      ],
      "should": [
        { "wildcard": { "구매사이트" : "11*"}},
        { "range": { "고객나이": { "lt": 30}}}
      ],
      "minimum_should_match": 1
    }
  }
}
```

## 5. A AND (B OR C OR D 중 2개 이상) 를 구해보자

A : 고객주소\_시도 = 서울특별시  
B : 구매사이트 = 11로 시작  
C : 고객나이 < 30  
D : 주문날짜 = 일요일

```
GET shopping/_search
{
  "query": {
    "bool": {
      "must": [
        { "term": { "고객주소_시도": "서울특별시" }},
      ],
      "should": [
        { "wildcard": { "구매사이트" : "11*"}},
        { "range": { "고객나이": { "lt": 30}}},
        { "script": { "script" : { "source" : "doc['주문시간'].date.dayOfWeek == 7"}}}
      ],
      "minimum_should_match": 2
    }
  }
}
```

아직 Bool Query가 익숙하지 않으면 우선 query string으로 시작하자  

(다만 기능이 제한적이다)

```
GET shopping/shopping/_search
{
  "query": {
    "query_string": {
      "query": "고객나이 : [10 TO 25] OR 구매사이트: 쿠팡",
      "analyze_wildcard": true
    }
  }
}
```

**꼭 모든 걸 Query DSL로 할 필요는 없다.**

**Search, Filter, Query DSL을 목적에 맞게 적절히 사용하자**

## shopping dashboard에서 아래 데이터만 표시하자 ↗

조건	내용
A	결제카드 = 시티
B	구매사이트가 [11번가, 쿠팡, 옥션] 중 1개
C	상품가격 $\geq$ 20,000
D	$35 \geq$ 고객나이 $\geq 27$
E	주문시간의 시간대가 18시 이후
F	고객주소_시도가 “경”으로 시작하는 3글자

1. A AND B 
2. C OR E 
3. (A OR D) AND NOT B 
4. (D OR E OR F) 중 2개 이상 
5. (D AND F) OR (A AND NOT E) 

질문 및 Feedback은

[gshock94@gmail.com](mailto:gshock94@gmail.com)로 주세요