

Elastic Stack 을 활용한 Data Dashboard 만들기

Week 4 - Elasticsearch API를 활용해보자



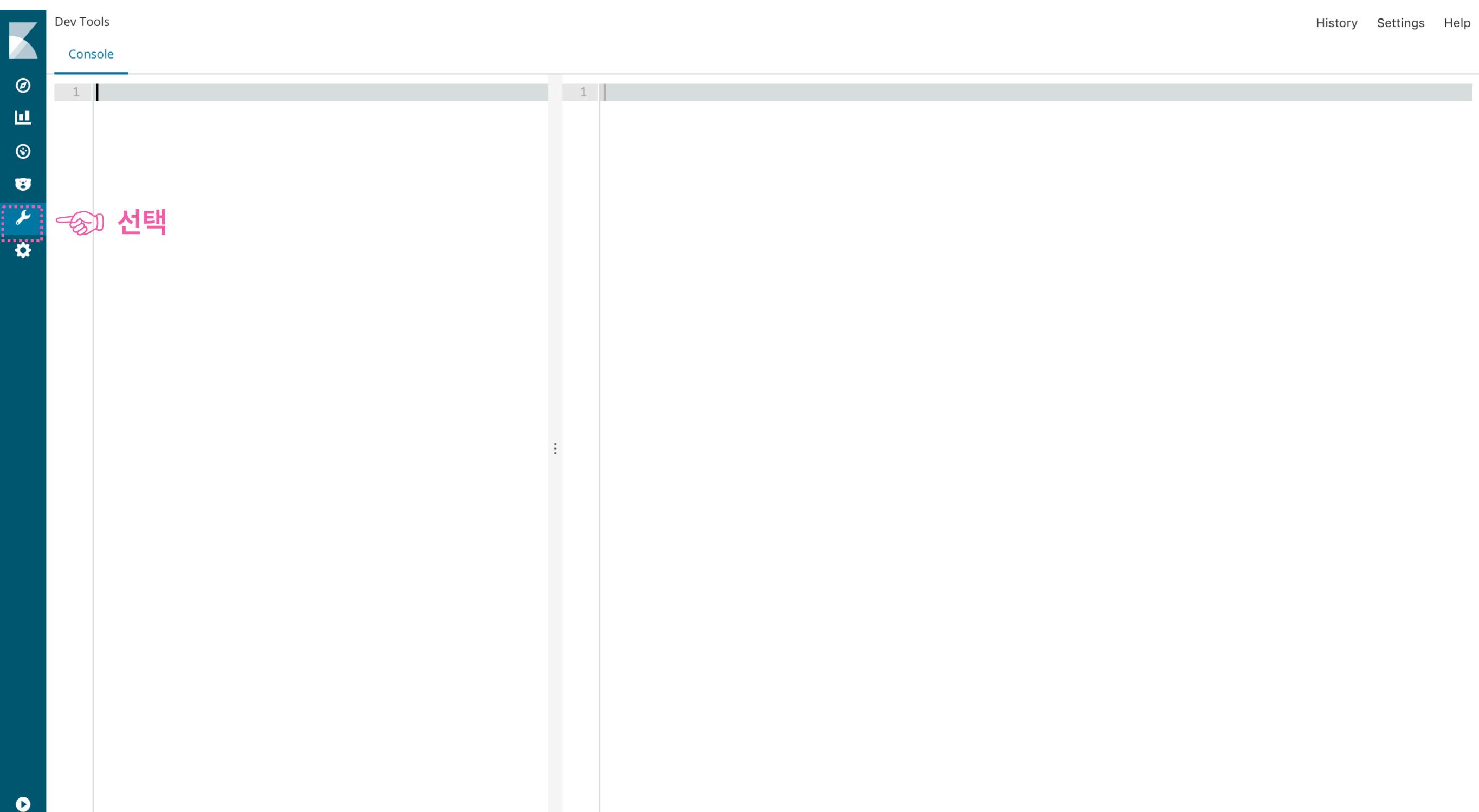
Fast Campus

내용	페이지
Dev Tools	3
Data Type	
Core datatype	11
Complex datatype	23
설치	48
API	
Indices API	
Create Index	75
Delete Index	76
Mapping	77
Document API	
Create Document	91
Get Document	93
Delete Document	94
Update Document	96
Reindex Document	100
Search API (Query DSL)	
Match All	111
Term/Terms	118
Match	120
Prefix/Wildcard/Fuzzy	136
Range	139
Exists	140
Query String	141
Bool	144

Kibana Dev Tools를 (간단히) 살펴보자

오늘 배우는 API는 (대부분) 여기에 작성한다

Kibana에 접속해서 Dev Tools 화면으로 가자



(우선) 다음과 같이 입력하고 녹색 버튼을 눌러보자

Dev Tools

Console

History Settings Help

1.  **입력**

2.  **선택**

GET /shopping/_search  

```
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": 20000,  
12    "max_score": 1,  
13    "hits": [  
14      {  
15        "_index": "shopping",  
16        "_type": "shopping",  
17        "_id": "opTsm2QByNsCKuKnwPrP",  
18        "_score": 1,  
19        "_source": {  
20          "접수 번호": 226,  
21          "주문 시간": "2018-09-17T09:03:11",  
22          "수령 시간": "2018-09-21T07:32:11",  
23          "예약 여부": "일반",  
24          "배송 메모": "환불 요청",  
25          "고객 ip": "81.67.231.56",  
26          "고객 성별": "남성",  
27          "고객 나이": 24,  
28          "물건 좌표": "36.155584374716945, 126.85527626055053",  
29          "고객 주소_시도": "충청북도",  
30          "구매 사이트": "g마켓",  
31          "판매 자평 점": 1,  
32          "상품 분류": "청바지",  
33          "상품 가격": 9000,  
34          "상품 개수": 1,  
35          "결제 카드": "우리"  
36        }  
37      }  
38    }  
39  }  
40 }  
41 }
```

cURL 명령어로 복사

Dev Tools

Console

1 GET /shopping/_search

Copy as cURL

Auto indent

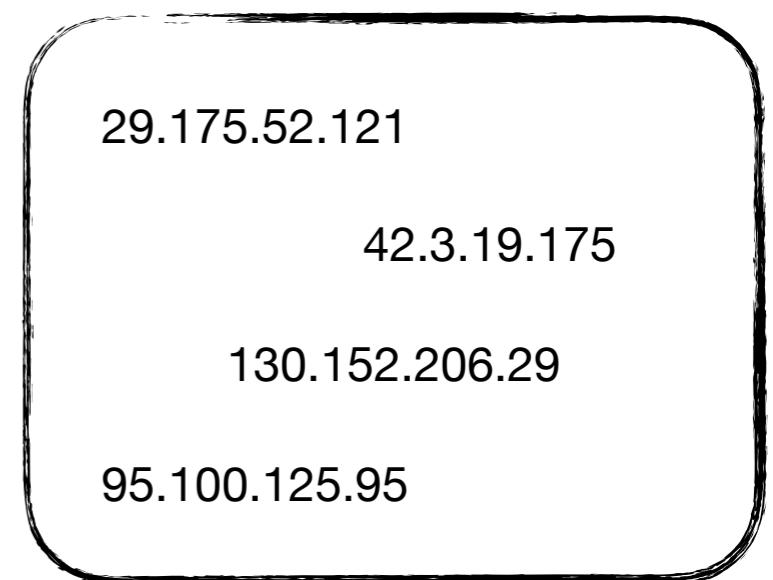
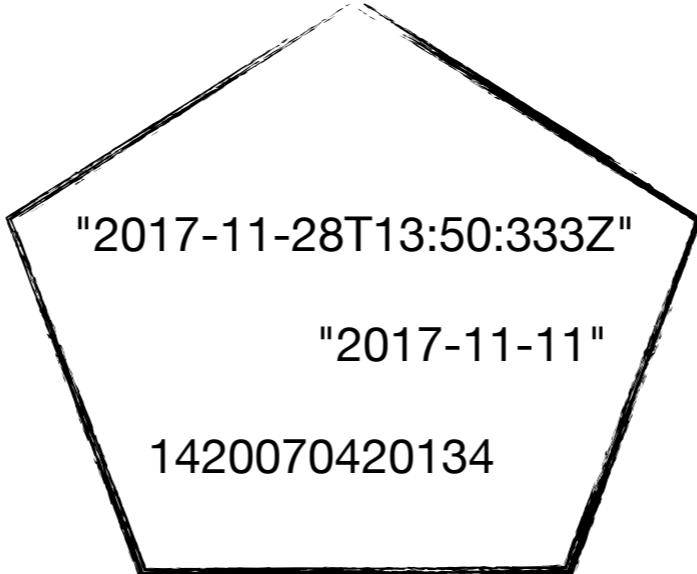
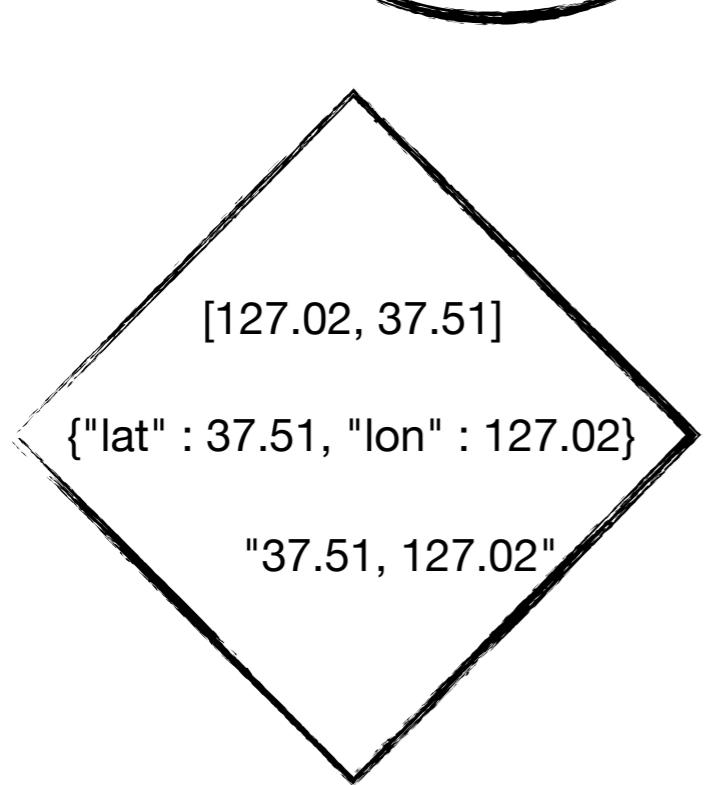
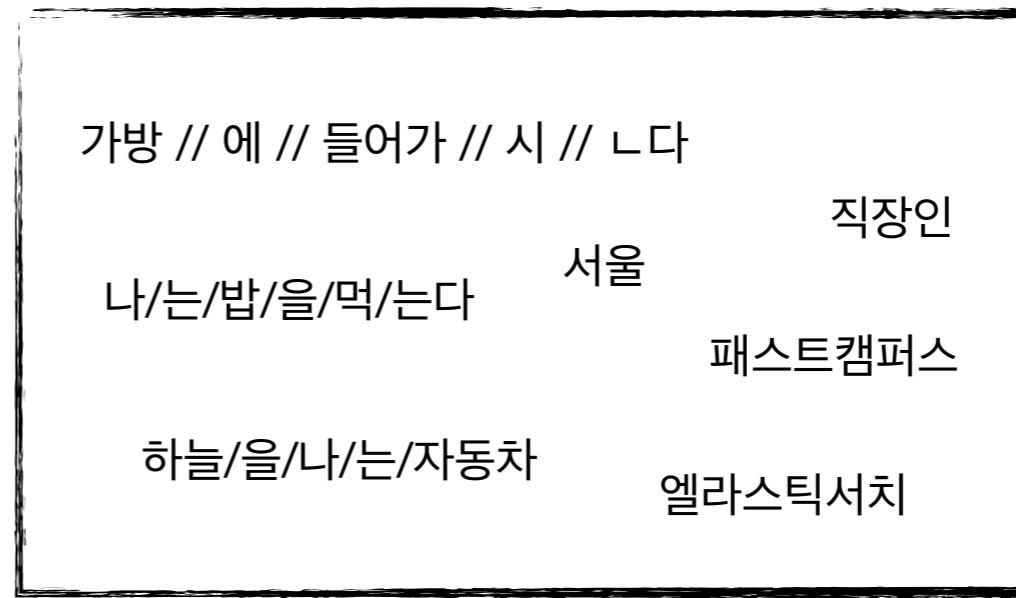
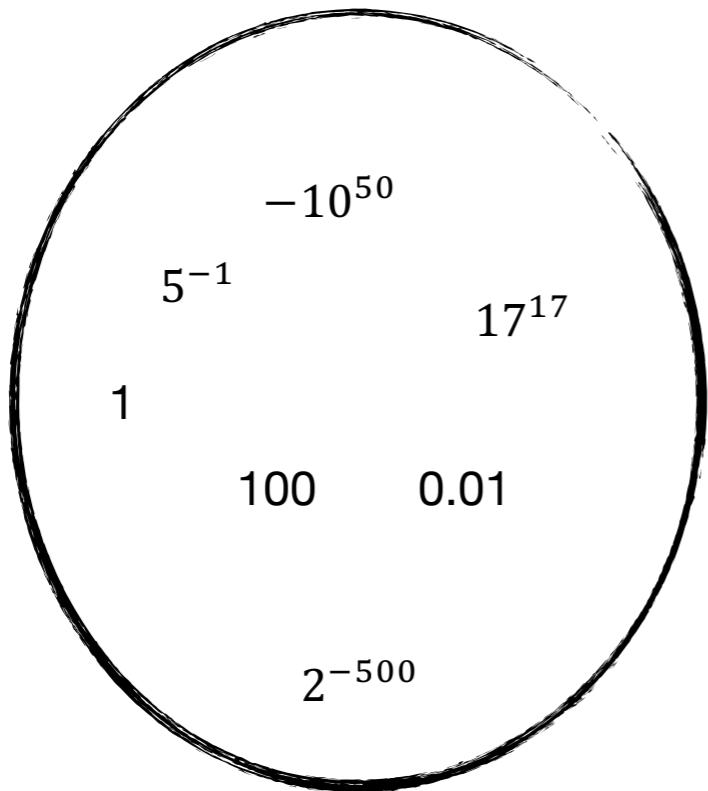
```
1 {  
2   "took": 0,  
3   "timed_out": false,  
4   "_shards": {  
5     "total": 5,  
6     "successful": 5,  
7     "skipped": 0,  
8     "failed": 0  
9   },  
10  "hits": {  
11    "total": 20000,  
12    "max_score": 1,  
13    "hits": [  
14      {  
15        "_index": "shopping",  
16        "_type": "shopping",  
17        "_id": "opTsm2OBvNsCKuKnwPrP",  
18        "_score": 1,  
19        "_source": {  
20          "접수번호": 226,  
21          "주문시간": "2018-09-17T09:03:11",  
22          "수령시간": "2018-09-21T07:32:11",  
23          "예약여부": "일반",  
24          "배송메모": "환불 요청",  
25          "고객ip": "81.67.231.56",  
26          "고객성별": "남성",  
27          "고객나이": 24,  
28          "물건좌표": "36.155584374716945, 126.85527626055053",  
29          "고객주소_시도": "충청북도",  
30          "구매사이트": "g마켓",  
31          "판매자평점": 1,  
32          "상품분류": "청바지",  
33          "상품가격": 9000,  
34          "상품개수": 1,  
35          "결제카드": "우리"  
36        }  
} }  
}
```

Console Elasticsearch API 작성

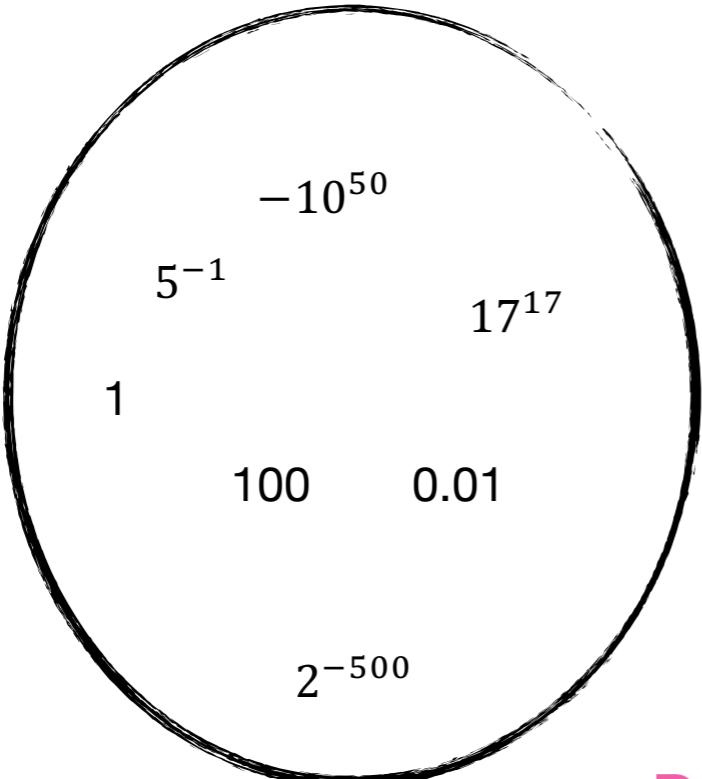
Output Pane 작성한 Elasticsearch API 결과 조회

kibana.higee.co/app/kibana#/home

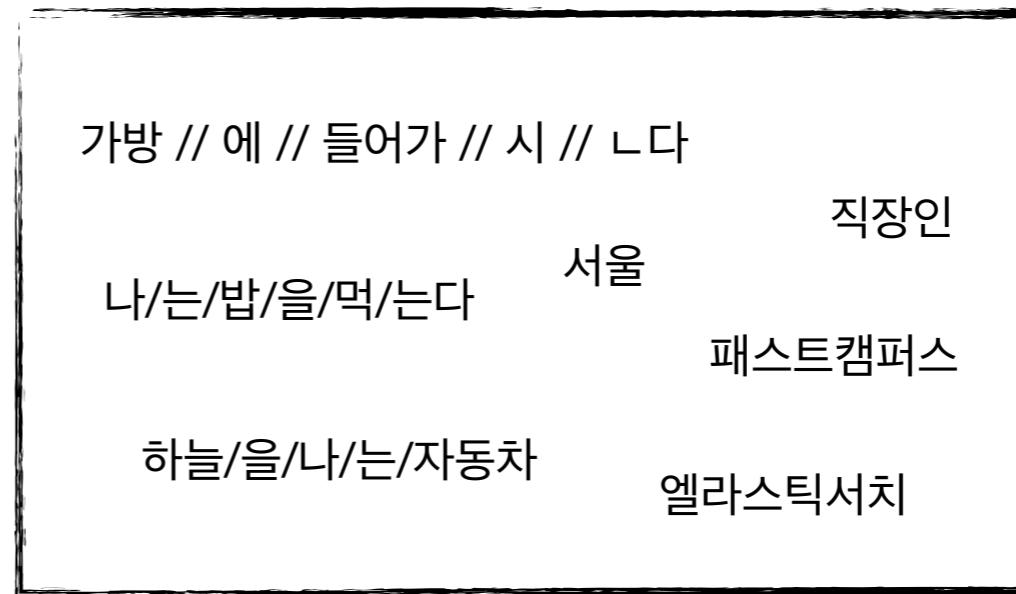
**Dashboard를 구축/운영 함에 있어
알면 좋은 (최소한의) datatype을 살펴보자**



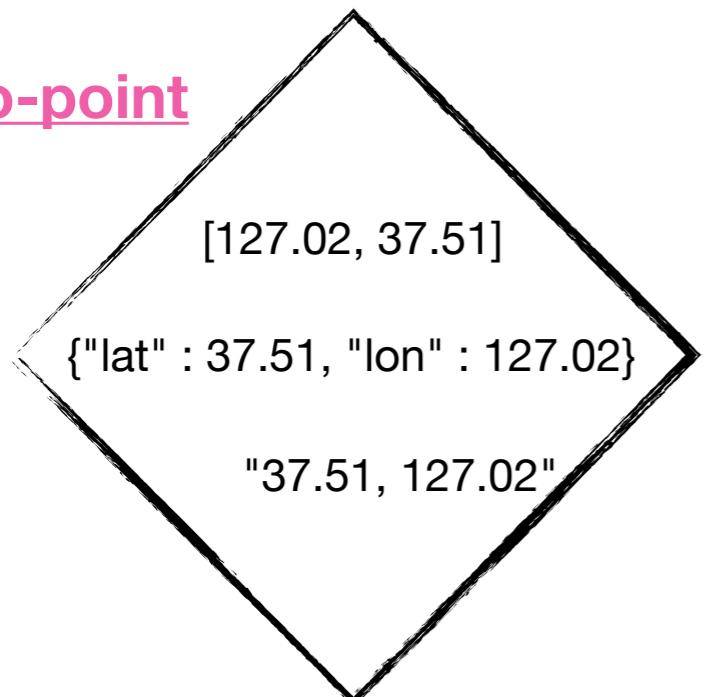
Numeric



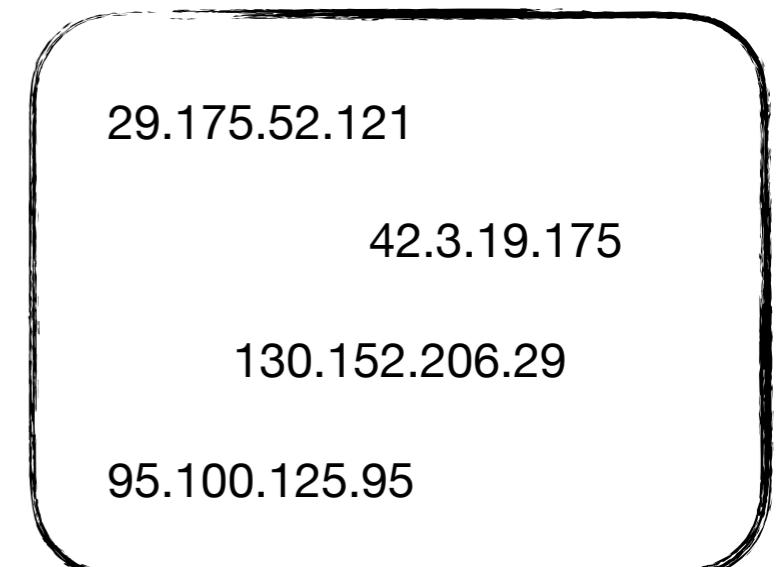
String



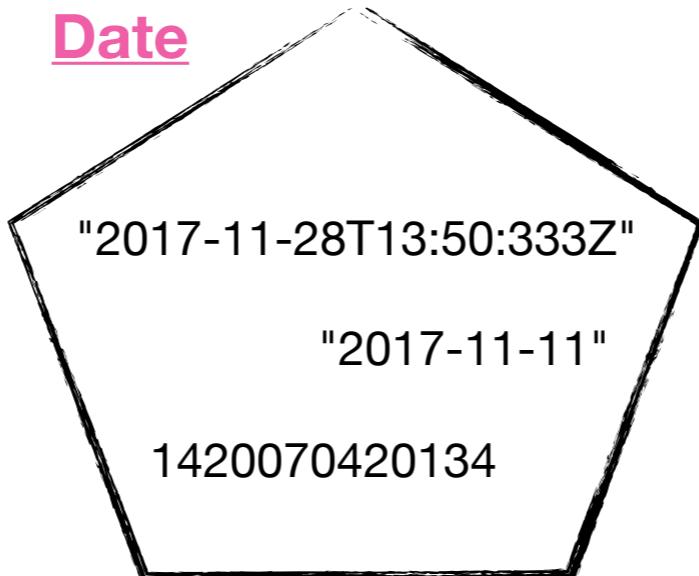
Date



IP



Geo-point



이 때 주의할 Type은 **Numeric**과 **String**

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

Numeric datatypes는 크게 정수형과 부동 소수점형으로 나눈다



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 .

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.

값의 범위

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, 낙낙한 (=안전하게) type을 사용하자

- 정수 (integer) : **Long**
- 부동소수점 (floating point number) : **Double**

정수형 (integer)

- (데이터를 담을 수 있는) Smallest Type 선택 : 검색/색인 성능 ↑
- 어떤 Type을 고르던 Storage 영향 ✘ : 실제 저장된 값의 크기에 따라 용량이 정해지기 때문
- 주의 : 실제 값을 담을 수 없는 Type을 선택하면 에러가 발생한다 (p 15)

부동소수점 (floating point number)

- (데이터 왜곡을 허용할 수 있는 범위 내의) Smallest Type 선택
- 어떤 Type을 사용하는지에 따라서 Storage 영향 ☑
- 주의 : 실제 값을 표현하기 부족한 Precision을 선택하면 예기치 않은 일이 생길 수 있다 (p 16)

Dev Tools

Console



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

POST my_index/my_type
{
  "test" : 129
}
```

The screenshot shows the Kibana Dev Tools interface with the 'Console' tab selected. On the left, there are icons for various tools: a magnifying glass, a bar chart, a clock, a dog, a wrench, and a gear. The main area displays two API requests. The first is a PUT request to 'my_index' with a mapping for 'my_type' where 'test' is defined as a 'byte'. The second is a POST request to 'my_index/my_type' with a document containing a single field 'test' set to the value 129. To the right of the requests, the response is shown:

```
{
  "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
}
```

	“test-double” field	“test-half-float” field
data type	double	half-float
indexing		3.1 3.111111 3.11111122222
search	GET /my_index/_search { "query" : { "range": { "test-double": { "gte": 3.111111, "lte" : 3.1111112 } } } }	GET my_index/_search { "query" : { "range": { "test-half-float": { "gte": 3.111111, "lte" : 3.1111112 } } } }
result	O	X

“test-double” field

“test-half-float” field

입력값

인식값

3.111111222:
test-double

3.111111222

3.111111:
test-double

3.111111

3.1: test-double

3.1

3.111111222:
test-half-float

3.111328125

3.111111:
test-half-float

3.111328125

3.1:
test-half-float

3.099609375



입력값 = 인식값

입력값 ≠ 인식값

**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) Keyword로 설정할 경우

2) Text로 설정 (분석기에 따라 상이)

가방에 들어가신다

"가방에 들어가신다"

"가방" // "에" // "들어가" // "시" // "ㄴ다"

나는 밥을 먹는다

"나는 밥을 먹는다"

"나" // "는" // "밥" // "을" // "먹" // "는다"

패스트캠퍼스 엘라스틱서치

"패스트캠퍼스 엘라스틱서치"

"패스트캠퍼스" // "엘라스틱서치"

자세히 살펴보자

	“test-text” field	“test-keyword” field
data type	text	keyword
indexing	“패스트캠퍼스 엘라스틱서치”	
search (=match query)	<pre>GET /my_index/_search { "query": { "term": { "test-text": "패스트캠퍼스" } } }</pre>	<pre>GET /my_index/_search { "query": { "term": { "test-keyword": "패스트캠퍼스" } } }</pre>
result	O	X

	"test-text" field	"test-keyword" field
data type	text	keyword
indexing	“패스트캠퍼스 엘라스틱서치”	“패스트캠퍼스 엘라스틱서치”
search (=match query)	<pre>GET /my_index/_search { "query": { "term": { "test-text": "패스트캠퍼스" } } }</pre>	<pre>GET /my_index/_search { "query": { "term": { "test-keyword": "패스트캠퍼스" } } }</pre>
result	O	X

자세한 건 뒤에서 배우고 keyword field와 text field의 차이 정도만 인식하고 넘어가자

~~API 학습 후에~~

조금 특별한 Data Type도 살펴보자 (= Complex datatypes)

데이터를 계층적으로 저장할 수 없을까? 

POST object/object

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

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"  
        }  
    ]  
}
```



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 (\neq array datatype)

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

실제 예시를 통해 비교해보자

nested 

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

array 

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

nested 

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

array 

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

이 때 nested/array index에 각각 아래와 같은 조건을 검색하면 어떻게 될까?

- 상품.구매사이트 = 11번가
• 상품.분류 = 셔츠

3.1 array datatype

심화

```
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"
            }
          ]
        }
      }
    ]
  }
}
```

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

검색된다 !!

왜 검색이 되지?

```
{
  "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": {
          "구매사이트": "11번가",
          "구매자": "민",
          "고객성별": "여",
          "상품": [
            {
              "구매사이트": "쿠팡",
              "분류": "셔츠"
            },
            {
              "구매사이트": "11번가",
              "분류": "팬츠"
            }
          ]
        }
      }
    ]
  }
}
```

```
"상품.구매사이트" : "쿠팡",  
"상품.분류" : "셔츠"
```

```
"상품.구매사이트" : "11번가",  
"상품.분류" : "팬츠"
```

원래 데이터는 위와 같은 두 object를 가진 array 형태였다.

"상품.구매사이트" : "쿠팡",
"상품.분류" : "셔츠"

"상품.구매사이트" : "11번가",
"상품.분류" : "팬츠"

Elasticsearch는 위의 Document가 아래의 조건을 만족한다고 판단한 것이다.



- 상품.구매사이트 = 11번가
- 상품.분류 = 셔츠

즉 array type을 사용하면,

Association을 무시하고

Elasticsearch는 위의 Document가 아래의 조건을 만족한다고 판단한 것이다.

단순히 value의 존재 유무만 고려한다

- 상품.구매사이트 = 11번가
- 상품.분류 = 셔츠

```
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": []
  }
}
```

3.2 nested datatype

심화

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": [  
            {  
              "구매 사이트": "쿠팡",  
              "분류": "셔츠"  
            },  
            {  
              "구매 사이트": "11번가",  
              "분류": "팬츠"  
            }  
          ]  
        }  
      }  
    ]  
  }  
}
```

3.2 nested datatype

심화

```
"상품.구매사이트" : "쿠팡",  
"상품.분류" : "셔츠"
```

```
"상품.구매사이트" : "11번가",  
"상품.분류" : "팬츠"
```

원래 데이터는 위와 같은 두 object를 가진 nested 형태였다.

"상품.구매사이트" : "쿠팡",
"상품.분류" : "셔츠"

"상품.구매사이트" : "11번가",
"상품.분류" : "팬츠"

상품.구매사이트와 상품.분류를 모두 이용해서 검색을 하려면 아래 조건 중에 하나로 해야 한다

- ∩
- 상품.구매사이트 = 11번가
 - 상품.분류 = 팬츠

- ∩
- 상품.구매사이트 = 쿠팡
 - 상품.분류 = 셔츠

"상품.구매사이트" : "쿠팡",
"상품.분류" : "셔츠"

"상품.구매사이트" : "11번가",
"상품.분류" : "팬츠"

즉, *Association*을 고려해서 *object* 단위에서

상품.구매사이트와 상품.분류를 모두 이용해서 검색을 하려면 아래 조건 중에 하나로 해야 한다

조건을 만족하는 걸 판별한 것이다.

- 상품.구매사이트 = 11번가
- 상품.분류 = 팬츠

- 상품.구매사이트 = 쿠팡
- 상품.분류 = 셔츠

**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](#)

[Install Elasticsearch with Debian Package](#)

[Install Elasticsearch with RPM](#)

[Install Elasticsearch with Windows MSI Installer](#)

[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

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

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

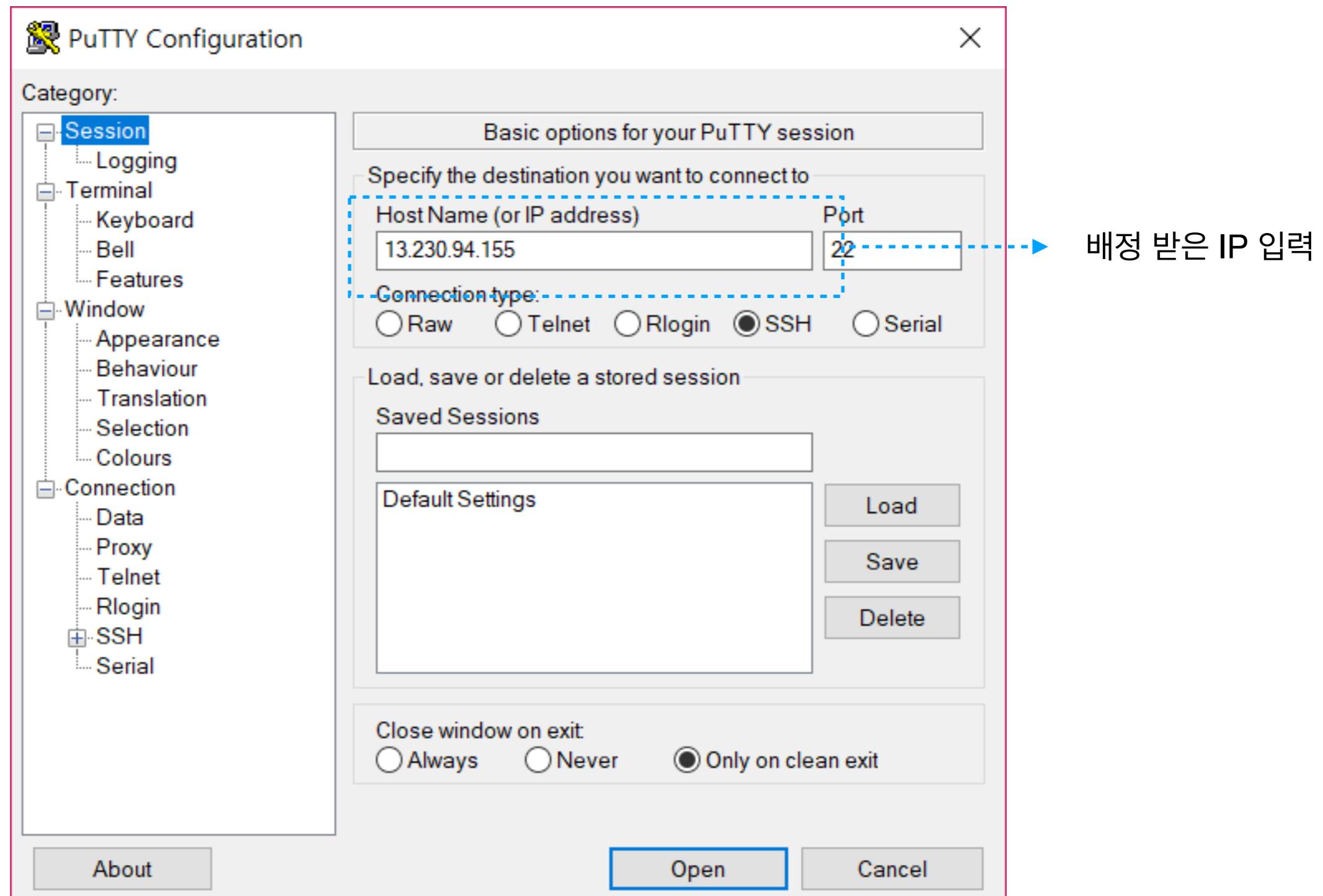
실제 업무 현장  에서 사용시에는 담당자들과 소통 후 환경을 설정하는 걸 권장한다.

설치 후 운영 중에도 끝없는 테스트가 수반되어야 한다.

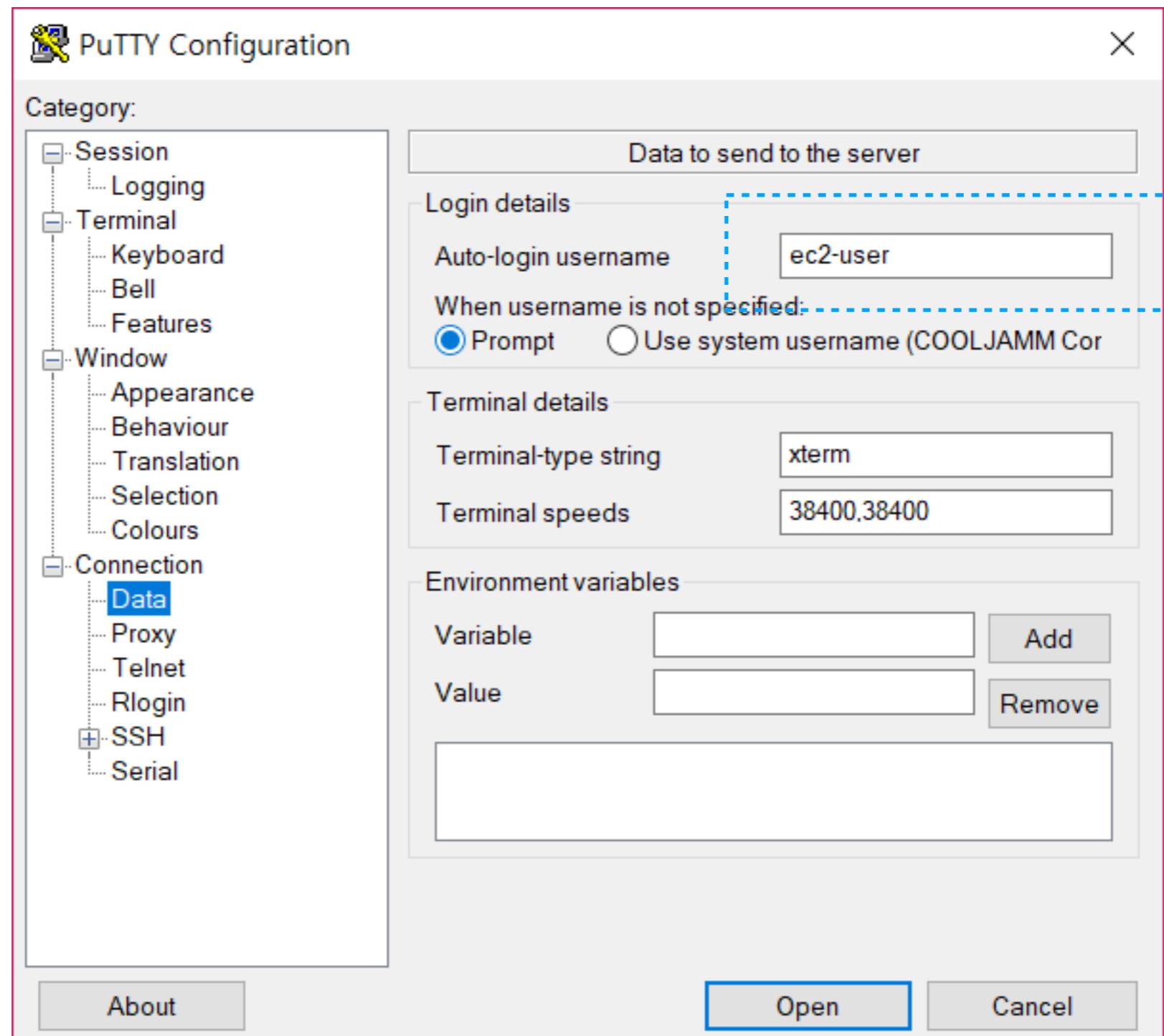
윈도우 - Putty 다운로드

- 32비트 : [클릭](#)
- 64비트 : [클릭](#)

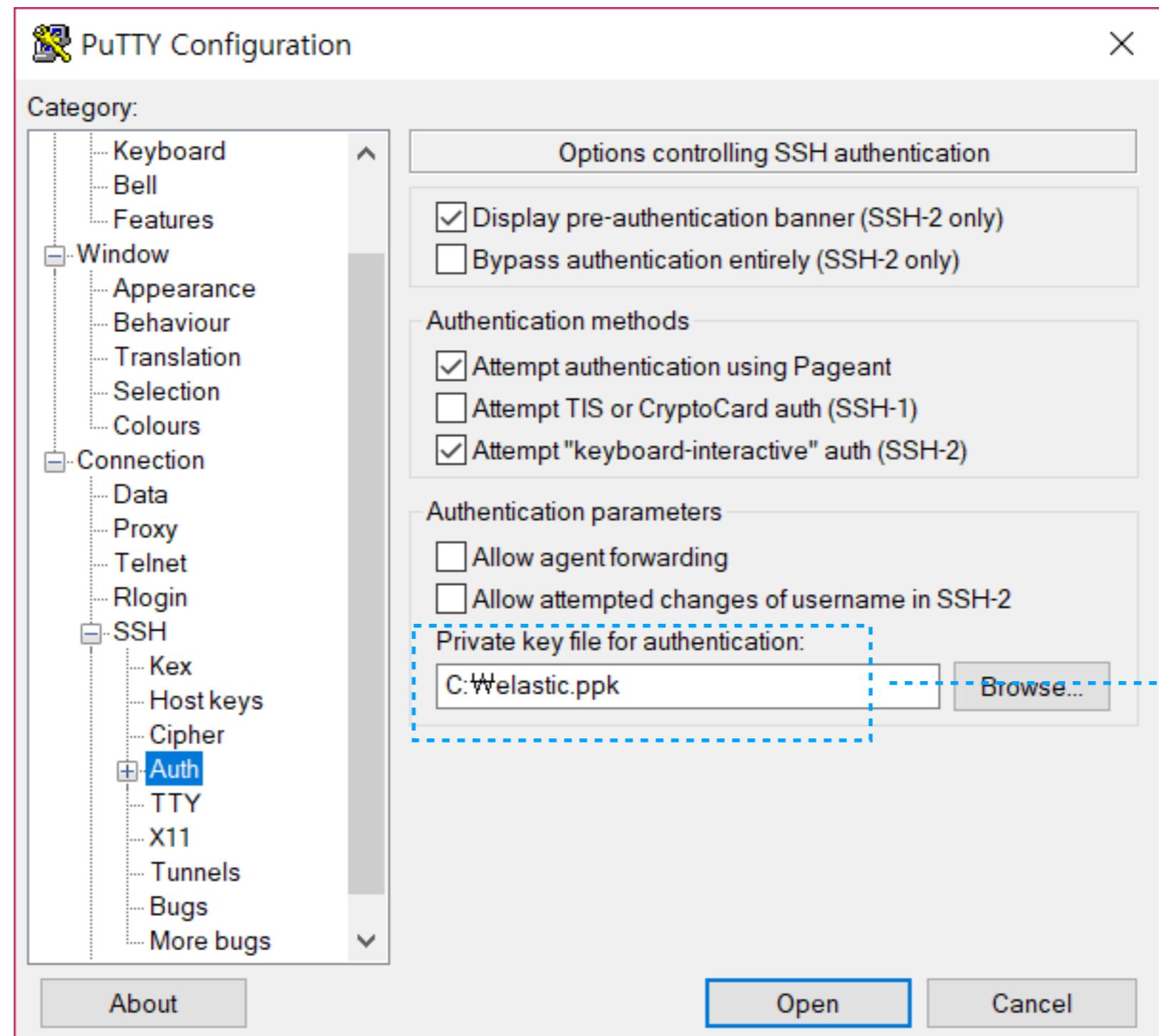
윈도우 - Putty 실행



윈도우 - Putty 실행

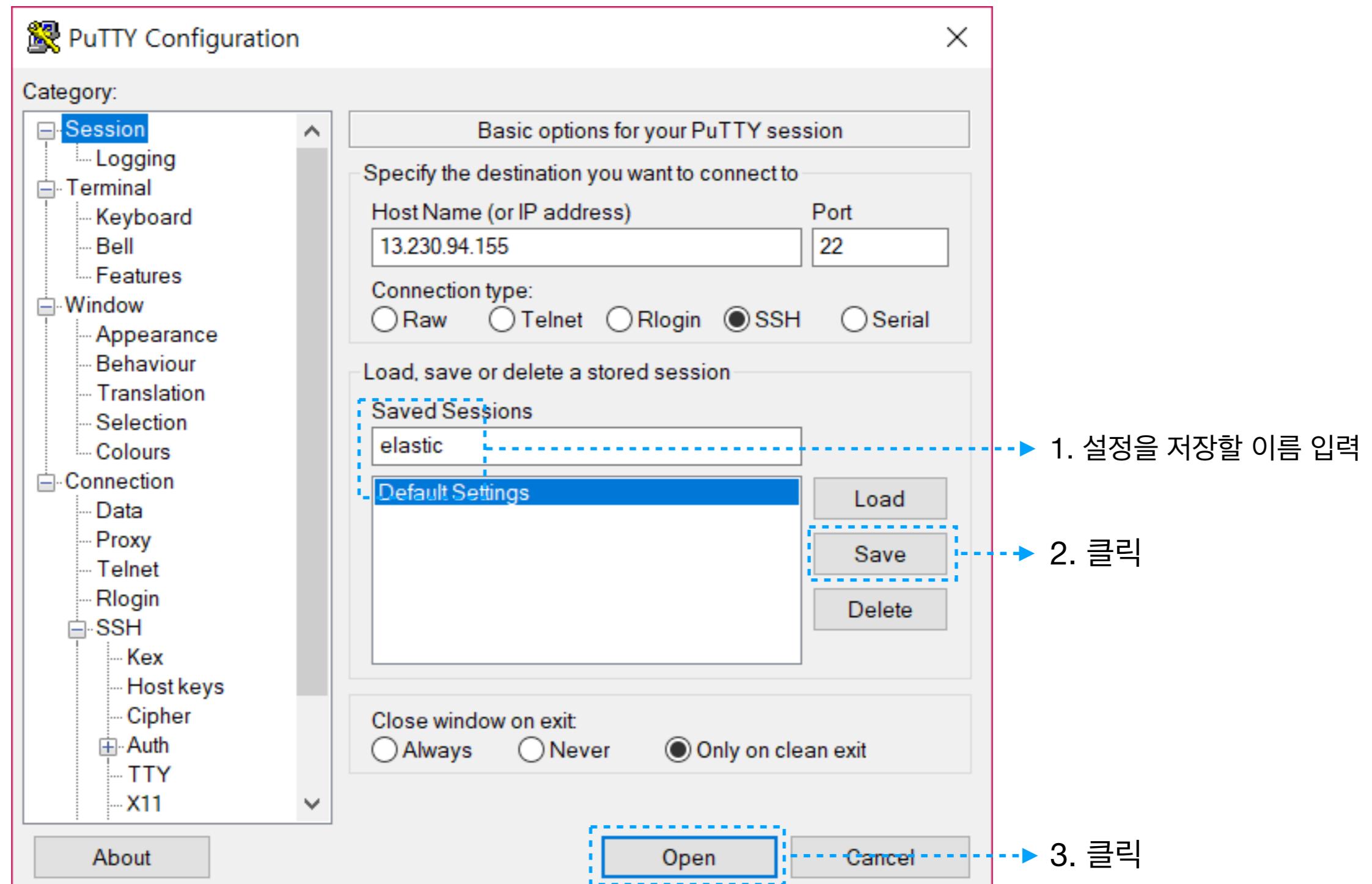


윈도우 - Putty 실행



사전에 드린
ppk 파일 선택

윈도우 - Putty 실행

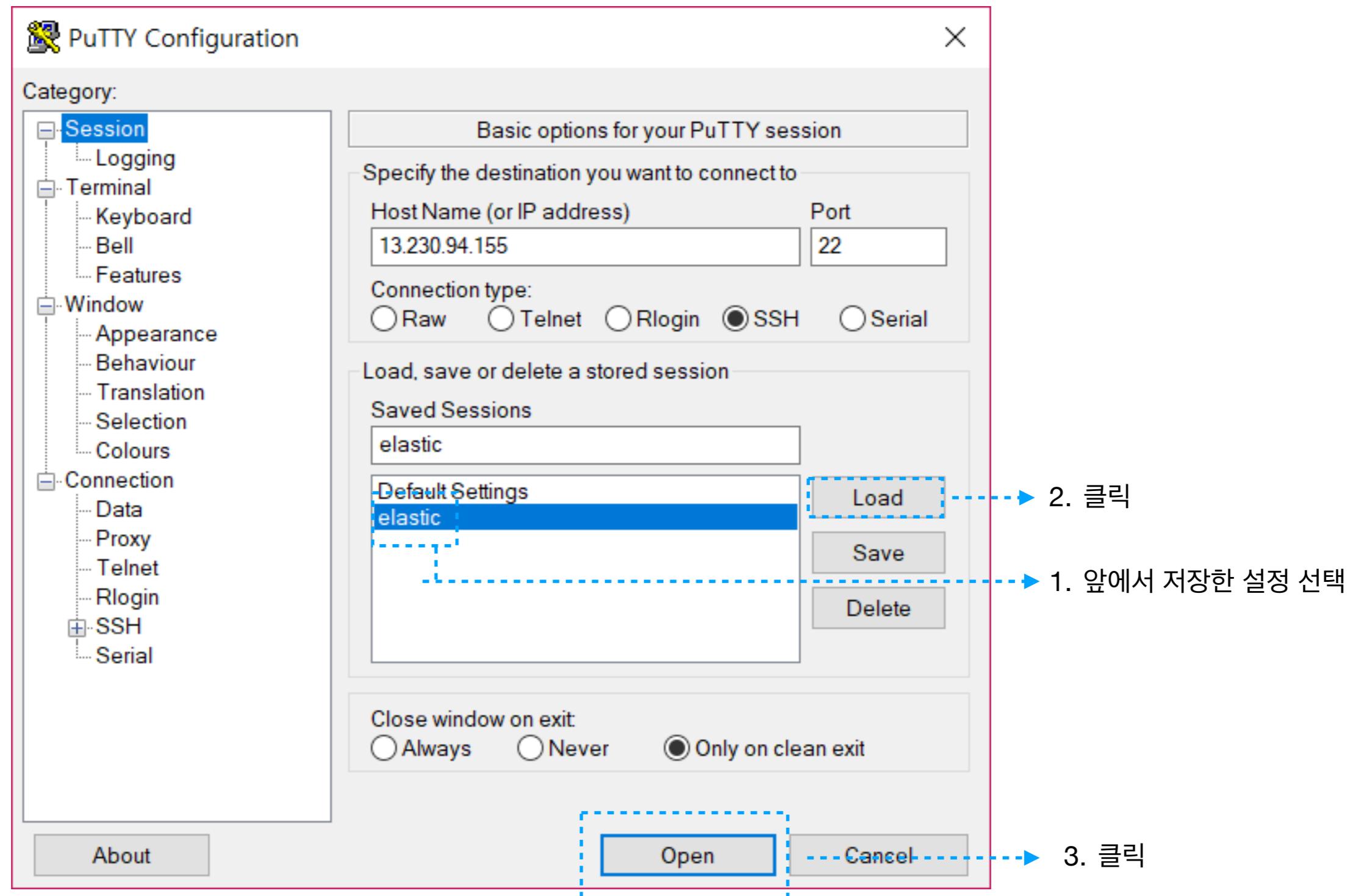


윈도우 - ec2 접속

The screenshot shows a Windows terminal window with a red border. The title bar reads "윈도우 - ec2 접속". The main area of the terminal displays a Linux command-line session:

```
ec2-user@ip-172-31-27-209:~  
Using username "ec2-user".  
Authenticating with public key "imported-openssh-key"  
Last login: Sun Sep 2 12:40:20 2018 from 118.221.38.242  
  
_ _ | _ _ | _ )  
_ | ( _ _ / Amazon Linux AMI  
_ | \ _ _ | _ |  
  
https://aws.amazon.com/amazon-linux-ami/2018.03-release-notes/  
8 package(s) needed for security, out of 13 available  
Run "sudo yum update" to apply all updates.  
[ec2-user@ip-172-31-27-209 ~]$
```

윈도우 - ec2 접속 (접속 종료 후 다시 접속 할 경우)



Mac OS - ec2 접속

```
$ ssh -it "elastic.pem" ec2-user@13.230.94.155
```

- 터미널을 열고 위 명령어를 입력하자
- 단,
 - ▶ 현재 경로에 elastic.pem이 있어야 하고
 - ▶ 접속하려는 IP 주소가 13.230.94.155이다

Mac OS - ec2 접속

```
gee@Gees-MacBook-Pro Downloads $ ssh -i "elastic.pem" ec2-user@13.230.94.155
Last login: Sun Sep  2 14:23:50 2018 from 118.221.38.242

 _ _|_ _|_) )
 _| ( / Amazon Linux AMI
 ___|\_\_|_\_|

https://aws.amazon.com/amazon-linux-ami/2018.03-release-notes/
8 package(s) needed for security, out of 13 available
Run "sudo yum update" to apply all updates.
-bash: warning: setlocale: LC_CTYPE: cannot change locale (UTF-8): No such file or directory
[ec2-user@ip-172-31-27-209 ~]$ █
```

Elastic Stack을 설치해보자 

주요 설정 (Linux 기준)

- Swap
- JVM Heap
- Maximum number of threads
- Maximum number of file descriptors
- Virtual Memory



Swapping Is the Death of Performance

It should be obvious, but it bears spelling out clearly: swapping main memory to disk will *crush* server performance. Think about it: an in-memory operation is one that needs to execute quickly.

If memory swaps to disk, a 100-microsecond operation becomes one that takes 10 milliseconds. Now repeat that increase in latency for all other 10us operations. It isn't difficult to see why swapping is terrible for performance.

The best thing to do is disable swap completely on your system. This can be done temporarily:

```
sudo swapoff -a
```

To disable it permanently, you'll likely need to edit your `/etc/fstab`. Consult the documentation for your OS.

If disabling swap completely is not an option, you can try to lower `swappiness`. This value controls how aggressively the OS tries to swap memory. This prevents swapping under normal circumstances, but still allows the OS to swap under emergency memory situations.

For most Linux systems, this is configured using the `sysctl` value:

```
vm.swappiness=1 ❶ ※ 0 ~ 100 사이 값을 가지며 0에 가까울 수록 swap 최대한 비활성화
```

- ❶ A `swappiness` of `1` is better than `0`, since on some kernel versions a `swappiness` of `0` can invoke the OOM-killer.

Finally, if neither approach is possible, you should enable `mlockall`. file. This allows the JVM to lock its memory and prevent it from being swapped by the OS. In your `elasticsearch.yml`, set this:

```
bootstrap.mlockall: true
```

주요 설정 - JVM Heap Memory

Heap: Sizing and Swapping



The default installation of Elasticsearch is configured with a 1 GB heap. For just about every deployment, this number is usually too small. If you are using the default heap values, your cluster is probably configured incorrectly.

There are two ways to change the heap size in Elasticsearch. The easiest is to set an environment variable called `ES_HEAP_SIZE`. When the server process starts, it will read this environment variable and set the heap accordingly. As an example, you can set it via the command line as follows:

```
export ES_HEAP_SIZE=10g
```

Alternatively, you can pass in the heap size via JVM flags when starting the process, if that is easier for your setup:

```
ES_JAVA_OPTS="-Xms10g -Xmx10g" ./bin/elasticsearch
```

- Ensure that the min (`Xms`) and max (`Xmx`) sizes are the same to prevent the heap from resizing at runtime, a very costly process.

1. min size = max size

Generally, setting the `ES_HEAP_SIZE` environment variable is preferred over setting explicit `-Xmx` and `-Xms` values.

Give (less than) Half Your Memory to Lucene



A common problem is configuring a heap that is *too* large. You have a 64 GB machine—and by golly, you want to give Elasticsearch all 64 GB of memory. More is better!

Heap is definitely important to Elasticsearch. It is used by many in-memory data structures to provide fast operation. But with that said, there is another major user of memory that is *off heap*: Lucene.

Lucene is designed to leverage the underlying OS for caching in-memory data structures. Lucene segments are stored in individual files. Because segments are immutable, these files never change. This makes them very cache friendly, and the underlying OS will happily keep hot segments resident in memory for faster access. These segments include both the inverted index (for fulltext search) and doc values (for aggregations).

Lucene's performance relies on this interaction with the OS. But if you give all available memory to Elasticsearch's heap, there won't be any left over for Lucene. This can seriously impact the performance.

The standard recommendation is to give 50% of the available memory to Elasticsearch heap, while leaving the other 50% free. It won't go unused; Lucene will happily gobble up whatever is left over.

If you are not aggregating on analyzed string fields (e.g. you won't be needing `fielddata`) you can consider lowering the heap even more. The smaller you can make the heap, the better performance you can expect from both Elasticsearch (faster GCs) and Lucene (more memory for caching).

2. 전체 메모리의 ½ 할당

Don't Cross 32 GB!



3. 최대 32GB 넘지 않도록

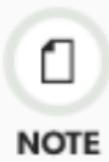
Maximum number of threads check



Elasticsearch executes requests by breaking the request down into stages and handing those stages off to different thread pool executors. There are different [thread pool executors](#) for a variety of tasks within Elasticsearch. Thus, Elasticsearch needs the ability to create a lot of threads. The maximum number of threads check ensures that the Elasticsearch process has the rights to create enough threads under normal use. This check is enforced only on Linux. If you are on Linux, to pass the maximum number of threads check, you must configure your system to allow the Elasticsearch process the ability to create at least 4096 threads. This can be done via

`/etc/security/limits.conf` using the `nproc` setting (note that you might have to increase the limits for the `root` user too).

File Descriptors



This is only relevant for Linux and macOS and can be safely ignored if running Elasticsearch on Windows. On Windows that JVM uses an [API](#) limited only by available resources.

Elasticsearch uses a lot of file descriptors or file handles. Running out of file descriptors can be disastrous and will most probably lead to data loss. Make sure to increase the limit on the number of open files descriptors for the user running Elasticsearch to 65,536 or higher.

For the `.zip` and `.tar.gz` packages, set `ulimit -n 65536` as root before starting Elasticsearch, or set `nofile` to `65536` in `/etc/security/limits.conf`.

On macOS, you must also pass the JVM option `-XX:-MaxFDLimit` to Elasticsearch in order for it to make use of the higher file descriptor limit.

RPM and Debian packages already default the maximum number of file descriptors to 65536 and do not require further configuration.



Virtual memory

Elasticsearch uses a `mmapfs` directory by default to store its indices. The default operating system limits on mmap counts is likely to be too low, which may result in out of memory exceptions.

On Linux, you can increase the limits by running the following command as `root`:

```
sysctl -w vm.max_map_count=262144 container 내부가 아닌 host에서 실행해야 한다 
```

To set this value permanently, update the `vm.max_map_count` setting in `/etc/sysctl.conf`. To verify after rebooting, run `sysctl vm.max_map_count`.

The RPM and Debian packages will configure this setting automatically. No further configuration is required.

`mmapfs`

The MMap FS type stores the shard index on the file system (maps to Lucene `MMapDirectory`) by mapping a file into memory (`mmap`). Memory mapping uses up a portion of the virtual memory address space in your process equal to the size of the file being mapped. Before using this class, be sure you have allowed plenty of `virtual address space`.

설치 - virtual memory

```
$ sudo vim /etc/sysctl.conf
```

```
# Controls the use of TCP syncookies  
net.ipv4.tcp_syncookies = 1  
  
# Controls the default maximum size of a message queue  
kernel.msgmnb = 65536  
  
# Controls the maximum size of a message, in bytes  
kernel.msgmax = 65536  
  
# Controls the maximum shared segment size, in bytes  
kernel.shmmax = 68719476736  
  
# Controls the maximum number of shared memory segments, in pages  
kernel.shmall = 4294967296
```

```
vm.max_map_count=262144
```

vm.max_map_count=262144 추가

```
$ sudo reboot
```

설치 - docker

EC2 접속 (Mac : ssh, Windows : putty)

```
$ sudo yum install docker -y  
$ sudo usermod -aG docker $USER  
$ exit
```

EC2 접속 (Mac : ssh, Windows : putty)

```
$ sudo service docker start
```

설치 - docker-compose

```
$ sudo curl \
-L https://github.com/docker/compose/releases/download/1.21.0/docker-compose-$(uname -s)-$(uname -m) \
-o /usr/local/bin/docker-compose
$ sudo chmod +x /usr/local/bin/docker-compose
```

설치 - git

```
$ sudo yum install git -y  
$ git clone -b class5 https://github.com/higee/elasticsearch.git  
$ cd /home/ec2-user/elasticsearch/Install/config
```

설치 - docker-compose 구동

```
$ sudo chown -R 1000:1000 /home/ec2-user/elastic/  
$ docker-compose up -d
```

docker-compose.yml

```
1  version: '3'  
2  
3  services:  
4  
5    elasticsearch:  
6      image: docker.elastic.co/elasticsearch/elasticsearch-oss:6.2.4  
7      container_name: elasticsearch  
8      environment:  
9        http.host: '0.0.0.0'  
10       network.host: '127.0.0.1'  
11       ES_JAVA_OPTS: '-Xms4g -Xmx4g -XX:-AssumeMP'  
12       bootstrap.memory_lock: 'true'  
13  
14     ulimits:  
15       memlock:  
16         soft: -1  
17         hard: -1  
18         -1 = unlimited  
19  
20      nofile:  
21         soft: 65536  
22         hard: 65536  
23       nproc:  
24         soft: 4096  
25         hard: 4096  
26  
27     ports:  
28       - "9200:9200"  
29       - "9300:9300"  
30  
31   restart: always  
32  
33   networks:  
34     - elastic
```

ES_JAVA_OPTS: '-Xms4g -Xmx4g -XX:-AssumeMP' → Heap Memory 설정 (p 62)

bootstrap.memory_lock: 'true' → swap 설정 (p 61)

soft: -1 → maximum number of file descriptors 설정 (p 64)

soft: 4096 → maximum number of threads 설정 (p 63)

다양한 Elasticsearch API 중에서

Indices, Document, Search API를 알아보자

Indices API로 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

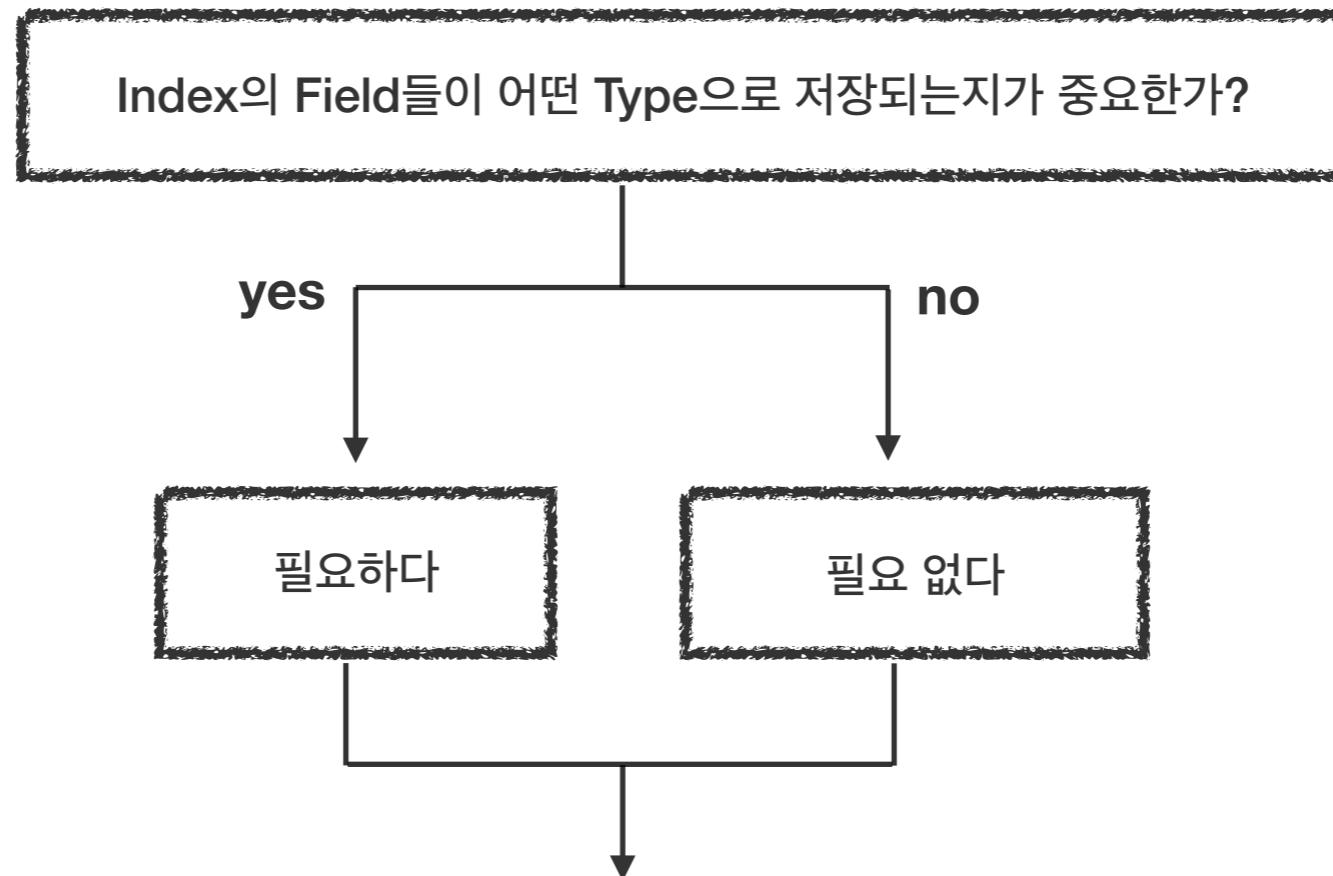
```
DELETE {Index 이름}
```

예시

```
DELETE week4_higee
```

Mapping API를 배우기 전에

Mapping 설정은 꼭 필요한가?



- 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

```
PUT {Index 이름}
```

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

예시

```
PUT week4_higee
```

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

Type 이름?? type?? 같은건가?

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

개별 Field의 Datatype

Index 하위 Type

두 개를 잘 구분하자!

Index 생성하면서 Mapping 추가

문법

```
DELETE {Index 이름}
```

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

예시

```
DELETE week4_higee
```

```
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"
    }
  }
}
```

문법

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

예시

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

**Template 생성 ≠ Index 생성
≠ Mapping 생성**



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

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

```
GET {Index 이름}/_mapping
```

예시

```
GET week4_higee/_mapping
```

Template Mapping 확인

문법

PUT {Index 이름}

예시

PUT higee-log-2018.01.01

GET {Index 이름}/_mapping

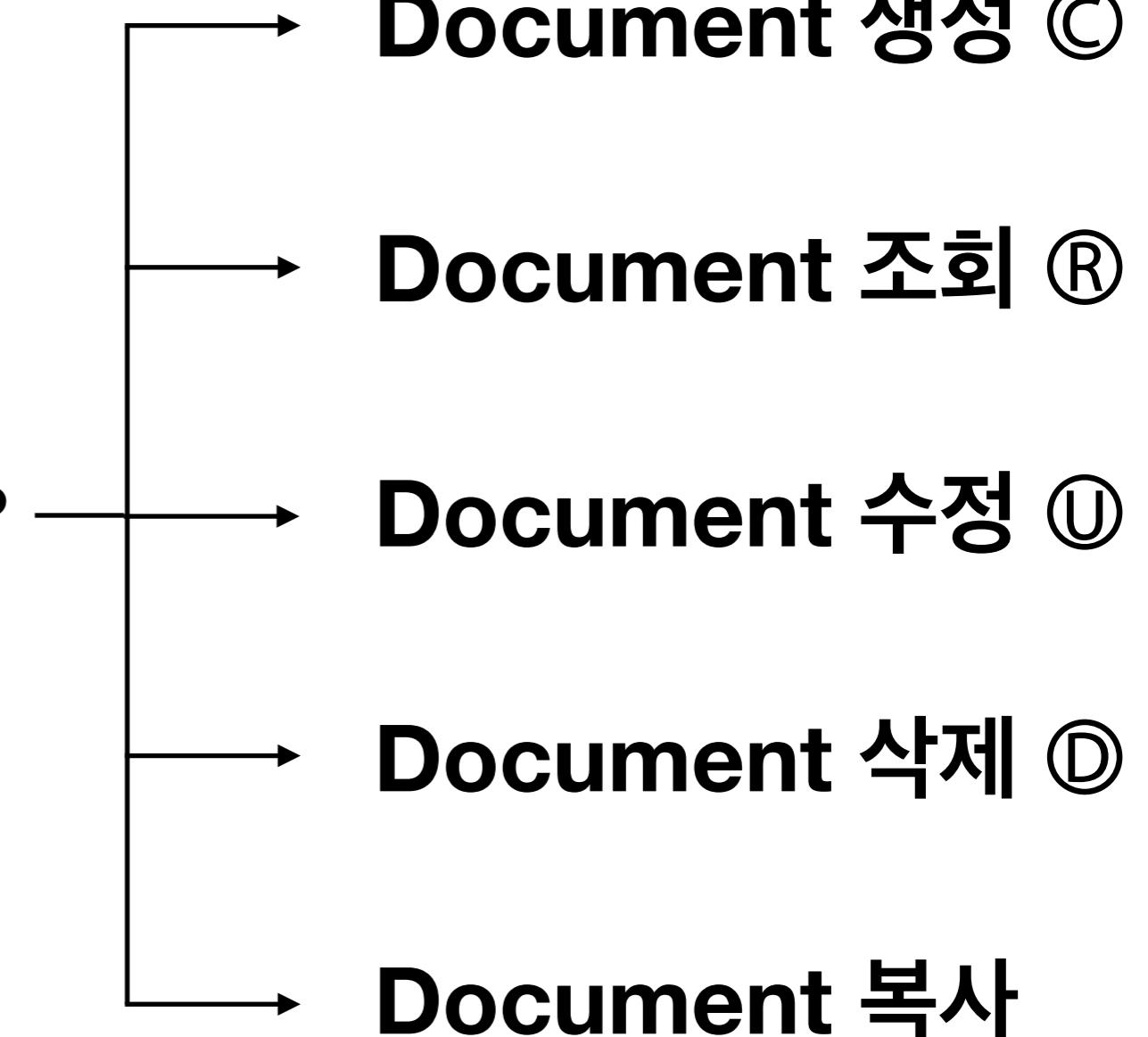
GET higee-log-2018.01.01/_mapping

예제1

1. 실습 서버의 shopping index의 mapping을 확인하고,
2. 동일한 index 이름, type 이름, mapping을 갖는 index를 자기 서버에 생성하자
3. 그리고 “환불여부”라는 Keyword Field를 추가하자

Document API로 무얼 할 수 있을까?

Document API로 무얼 할 수 있을까?



Document 추가 (지정 ID)

문법	예시
<pre>PUT {Index 이름}/{Type 이름}/{ID} { "{Field 이름}" : {Value} }</pre>	<pre>PUT week4_higee/week4_higee/1 { "price" : 10000, "age" : 17 }</pre>
	<pre>PUT week4_higee/week4_higee/2 { "price" : 2000, "age" : 20 }</pre>
	<pre>PUT week4_higee/week4_higee/3 { "price" : 1000, "age" : 25 }</pre>
	<pre>PUT week4_higee/week4_higee/4 { "price" : 7000, "age" : 33 }</pre>

Document 추가 (임의 ID)

문법

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

예시

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

ID로 Document 조회

문법

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

예시

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

ID로 Document 삭제

문법

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

예시

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

Query로 Document 삭제

문법

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

예시

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

ID로 Document 부분 수정

문법

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

예시

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

ID로 Document 전체 수정

문법

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

예시

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

ID로 Document 설정 (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
<pre>POST {Index 이름}/{Type 이름}/_update_by_query { "script": { "source": "ctx._source[{Field 이름}] = Value" }, "query": { "term": { "{Field 이름)": "Value" } } }</pre>	<pre>POST week4_higee/week4_higee/_update_by_query { "script": { "source": "ctx._source['age'] = 50" }, "query": { "term": { "age": 33 } } }</pre>
	<pre>POST week4_higee/week4_higee/_update_by_query { "script": { "source": "ctx._source.age = 70" }, "query": { "term": { "age": 50 } } }</pre>

Index 내 모든 Document 재색인

문법

```
POST _reindex
{
  "source": {
    "index": "{재색인하려는 원본 Index 이름}"
  },
  "dest": {
    "index": "{재색인 후 저장할 Index 이름}"
  }
}
```

예시

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

위와 같이 기본 옵션으로 사용할 경우 단순히 **Documents** 복사라고 볼 수 있다

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 권장한다

Index 내 일부 Document 재색인

문법

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

예시

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

1. **exercise2-without-mapping** index 생성
2. 다음과 같은 Document 생성 (type 이름 : **exercise2**)

```
"name" : "elastic stack",
"major_version" : 6,
"version" : "6.2.4"
```

3. **exercise2-without-mapping** index의 mapping 확인
4. index 이름이 **exercise2-with-mapping**, type 이름이 **exercise2**인 index를 만들면서 아래와 같은 mapping 생성

```
"name" : "keyword",
"major_version" : "byte"
"version" : "keyword"
```

5. **exercise2-without-mapping** (source)를 **exercise2-with-mapping** (destination)로 재색인
6. **exercise2-with-mapping**의 mapping 확인

1. **exercise2-without-mapping** index 생성
2. 다음과 같은 Document 생성 (type 이름 : **exercise2**)

```
"name" : "elastic stack",
"major_version" : 6,
"version" : "6.2.4"
```

Mapping은 수정이 안되므로

3. **exercise2-without-mapping** index의 mapping 확인
4. index 이름이 **exercise2-without-mapping** type 이름이 **exercise2**인 index를 만들면서 아래와 같은 mapping 생성
이와 같은 방법으로 원하는 Data type으로 바꿀 수 있다

```
"name" : "keyword",
"major_version" : "byte",
"version" : "keyword"
```

5. **exercise2-without-mapping**을 **exercise2-with-mapping**로 재색인
6. **exercise2-with-mapping**의 mapping 확인

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



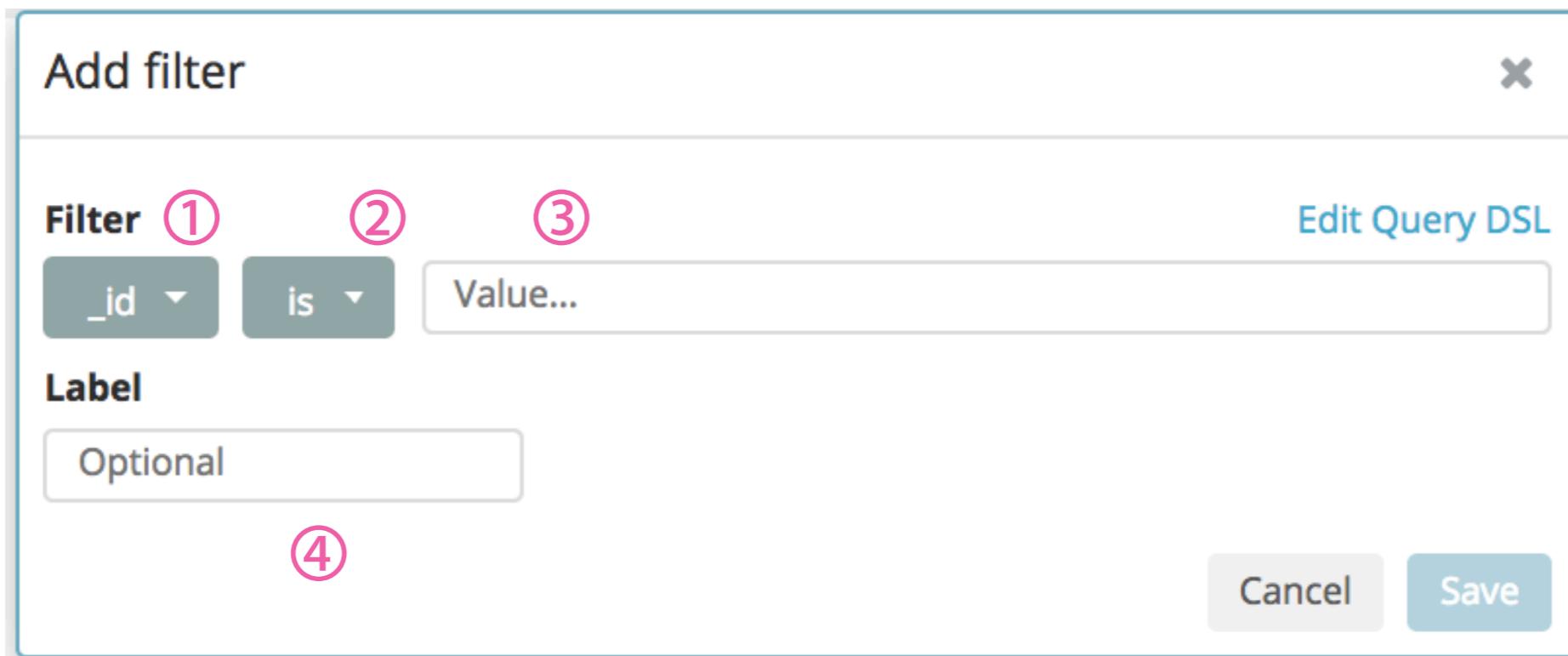
Request Body Search에서 사용되는 Domain Specific Language

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?

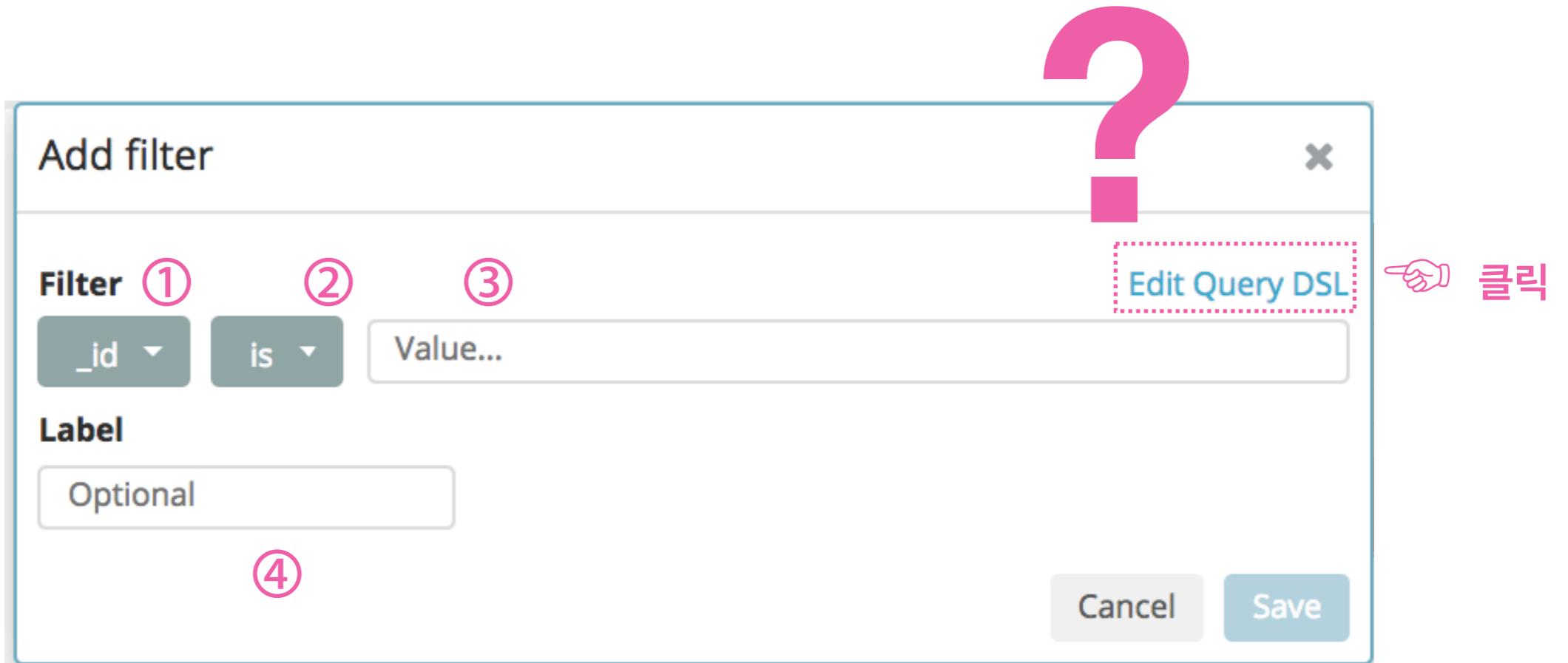
Filter 기능 강화를 위해서

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       "nginx.access.response_code" : "200"  
5     }  
6   }  
7 }
```

Search filter values



1. 입력

Filters are built using the Elasticsearch Query DSL.

Label

정상



2. 입력

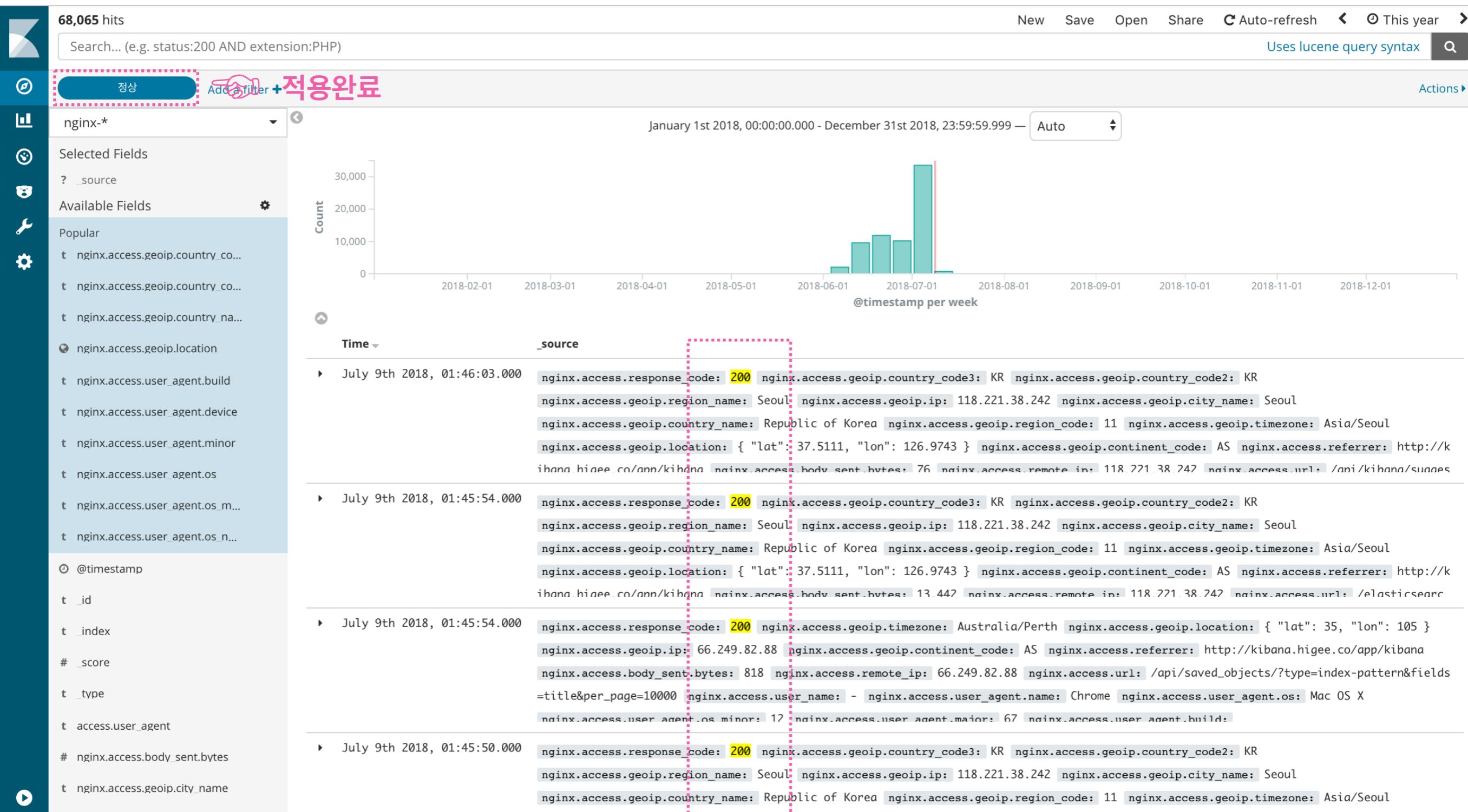
Cancel

Save



3. 선택

이런 결과가 나온다



데이터 확인

즉, Query DSL을 이용하면

AND 연산
OR 연산
Scripted Field
Wildcard 검색
Fuzzy/Proximity 검색



문제	Filter	Search	Query DSL
nginx.access.response_code가 200인 Doc	✓	✓	✓
nginx.access.method가 GET 또는 POST인 Doc	✓	✓	✓
nginx.access.geoip.region_name가 non-null값만 가지는 Doc	✓	✓	✓
nginx.access.geoip.city_name이 Seoul이면서 nginx.access.user_agent.name이 Chrome인 Doc	✓	✓	✓
nginx.access.geoip.city_name이 Seoul이거나 nginx.access.user_agent.name이 Chrome인 Doc		✓	✓
요일_local이 Sunday인 Doc 필터링	✓		✓
nginx.access.geoip.country_name 가 Republic of로 시작하는 Doc		✓	✓
nginx.access.geoip.continent_code가 AS와 유사한 Doc		✓	✓

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": {
④    "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

문법

예시

GET /{Index_이름}/{Type_이름}/search
{
 "query" : {
 "match_all" : {}
 }
}
Elasticsearch 6.0 이후에는 Index 별로 Type이 하나씩만 생긴다,
즉, 검색할 때도 Type을 생략해도 같은 결과가 나온다.
그러므로 나머지 실습에서는 Type은 생략하도록 한다

모든 Documents 조회

문법

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

예시

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

문법

```
GET /{Index 이름}/_search
{
  "query" : {
    "match_all" : {}
  },
  "sort": [
    {
      "{Field 이름)": {
        "order": "{desc 또는 asc}"
      }
    }
  ]
}
```

예시

```
GET /shopping/_search
{
  "query" : {
    "match_all" : {}
  },
  "sort": [
    {
      "판매자평점": {
        "order": "desc"
      }
    }
  ]
}
```

문법

```
GET /{Index 이름}/_search
{
  "from" : {FROM},
  "size" : {SIZE},
  "query" : {
    "match_all" : {}
  }
}
```

{FROM} 번째 Documents부터 {SIZE 개 }를 보여준다

예시

```
GET /shopping/_search
{
  "from" : 0,
  "size" : 1,
  "query" : {
    "match_all" : {}
  }
}
```

첫번째 Documents를 보여준다

단, size는 10,000개가 max이고 그 이상은 scroll()을 이용해서 구현() 해야 한다

문법

```
GET /{Index 이름}/_search
{
  "_source" : ["Field1", ...],
  "query" : {
    "match_all" : {}
  }
}
```

예시

- 기본

```
GET /shopping/_search
{
  "_source" : ["구매사이트"],
  "query" : {
    "match_all" : {}
  }
}
```

- includes/excludes 옵션 사용

```
GET /shopping/_search
{
  "_source": {
    "includes" : ["고객*", "구매사이트"],
    "excludes" : "상품*"
  },
  "query" : {
    "match_all" : {}
  }
}
```

검색어와 정확히 일치하는 value를 가진 Document 조회

문법

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

예시

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

검색어 중 적어도 1개와 정확히 일치하는 Document 조회

문법

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

예시

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

검색어와 부분적으로 일치하는 value를 가진 Document 조회

문법

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

예시

- “배송 못함”

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

- “시간 못함”

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

잠깐, Match Query = Term Query ?

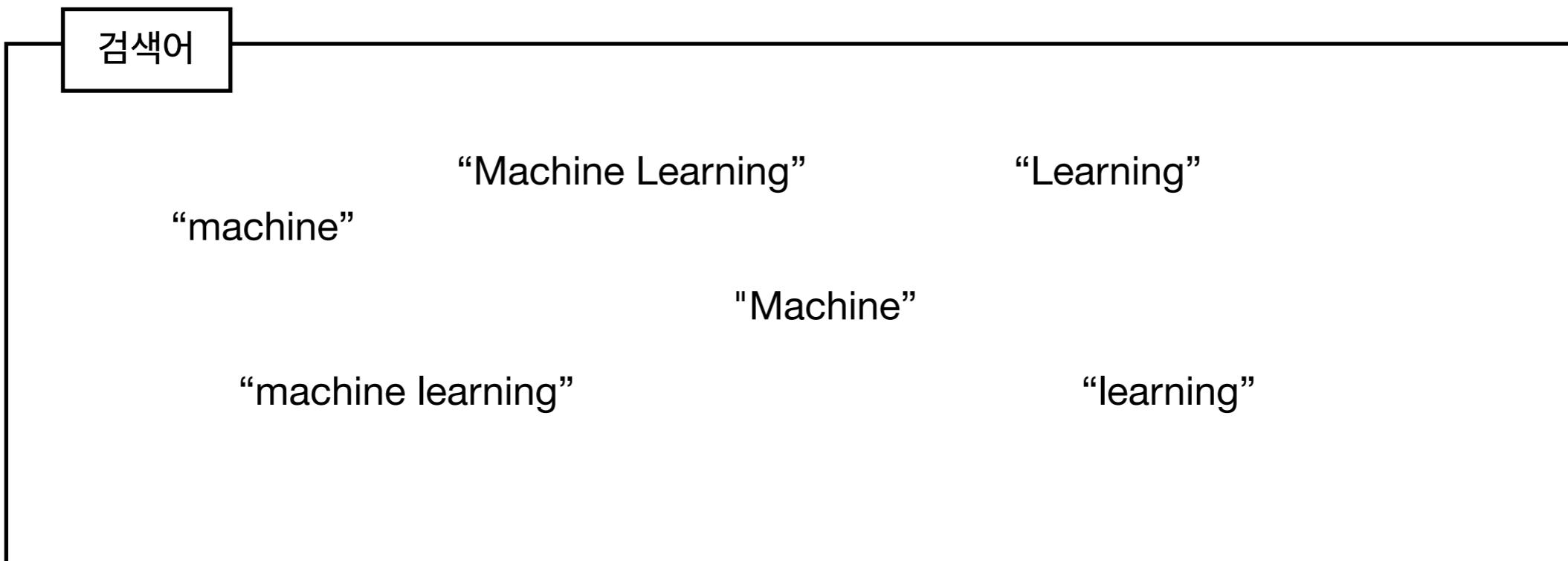
당연히 아니다

다만, 여기서는 개념적으로 어떻게 다른지 감만 잡고 가자

(**full text search**에 관심이 있으면 잘 알아야 한다)

“Machine Learning”이라는 데이터를 색인하고 아래와 같은 검색어로 검색해보자

심화



검색어

어떻게 될까?

“machine”

“Machine”

“machine learning”

“learning”

mapping : keyword 또는 text

analyzer : standard analyzer (=whitespace 구분 및 lowercase 적용)

query : term 또는 match

색인

색인 데이터	매핑	분석기
Machine Learning	keyword	standard analyzer
text		

검색

쿼리	검색어	결과
term	machine	x
term	Machine	x
term	machine learning	x
term	Machine Learning	o
match	machine	x
match	Machine	x
match	machine learning	x
match	Machine Learning	o
term	machine	x
term	Machine	x
term	machine learning	x
term	Machine Learning	o
match	machine	x
match	Machine	x
match	machine learning	x
match	Machine Learning	o
term	machine	x
term	Machine	x
term	machine learning	x
term	Machine Learning	o

- 매팅 

```
PUT ml
{
  "mappings": {
    "doc" : {
      "properties" : {
        "keyword" : {
          "type" : "keyword"
        },
        "text" : {
          "type" : "text"
        }
      }
    }
  }
}
```

- 색인 

```
POST ml/doc/
{
  "keyword" : "Machine Learning",
  "text" : "Machine Learning"
}
```

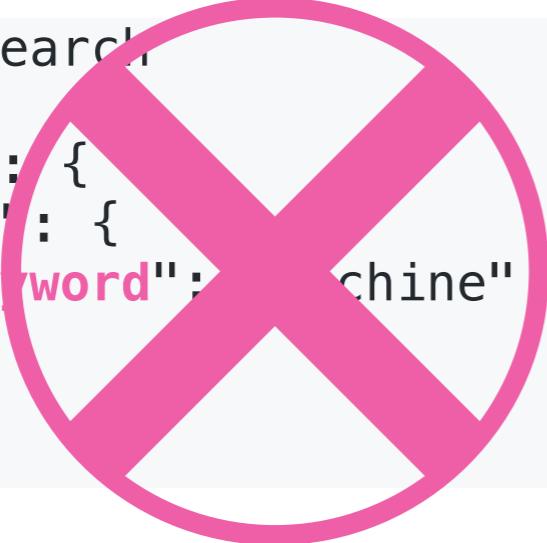
```
GET ml/_search
{
  "query": {
    "term": {
      "keyword": "machine"
    }
  }
}
```

```
GET ml/_search
{
  "query": {
    "term": {
      "keyword": "Machine"
    }
  }
}
```

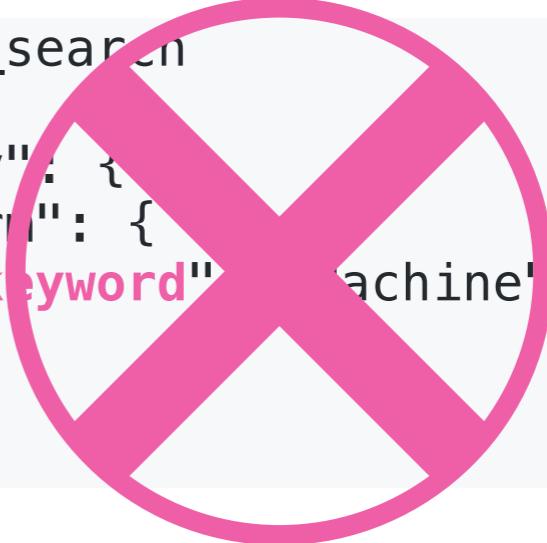
```
GET ml/_search
{
  "query": {
    "term": {
      "keyword": "machine learning"
    }
  }
}
```

```
GET ml/_search
{
  "query": {
    "term": {
      "keyword": "Machine Learning"
    }
  }
}
```

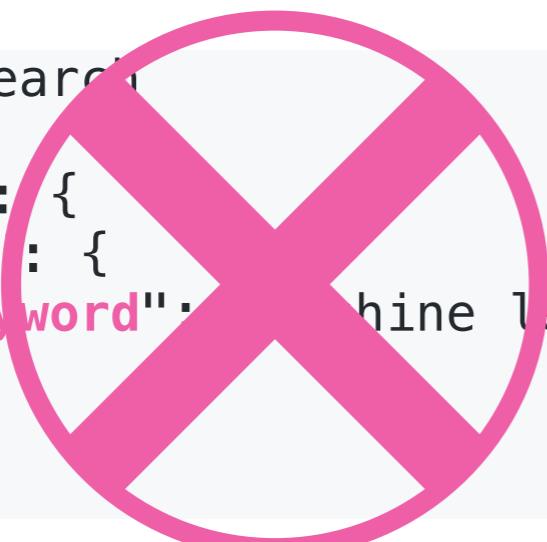
```
GET ml/_search
{
  "query": {
    "term": {
      "keyword": "Machine"
    }
  }
}
```



```
GET ml/_search
{
  "query": {
    "term": {
      "keyword": "Machine"
    }
  }
}
```



```
GET ml/_search
{
  "query": {
    "term": {
      "keyword": "Machine learning"
    }
  }
}
```



```
GET ml/_search
{
  "query": {
    "term": {
      "keyword": "Machine Learning"
    }
  }
}
```

keyword field에 term query를 사용하면 정확히 일치하는 것만 검색된다

```
GET ml/_search
{
  "query": {
    "match": {
      "keyword": "machine"
    }
  }
}
```

```
GET ml/_search
{
  "query": {
    "match": {
      "keyword": "Machine"
    }
  }
}
```

```
GET ml/_search
{
  "query": {
    "match": {
      "keyword": "machine learning"
    }
  }
}
```

```
GET ml/_search
{
  "query": {
    "match": {
      "keyword": "Machine Learning"
    }
  }
}
```

```
GET ml/_search  
{  
  "query": {  
    "match": {  
      "keyword": "Machine"  
    }  
  }  
}
```

```
GET ml/_search  
{  
  "query": {  
    "match": {  
      "keyword": "Machine"  
    }  
  }  
}
```

```
GET ml/_search  
{  
  "query": {  
    "match": {  
      "keyword": "Machine learning"  
    }  
  }  
}
```

```
GET ml/_search  
{  
  "query": {  
    "match": {  
      "keyword": "Machine Learning"  
    }  
  }  
}
```

match query를 사용하더라도 **keyword field**에 대한 검색이기에 정확히 일치하는 것만 검색된다.
즉, 검색 수행 시 검색어에 대한 **analyze**를 거치지 않는다.

```
GET ml/_search
{
  "query": {
    "term": {
      "text": "machine"
    }
  }
}
```

```
GET ml/_search
{
  "query": {
    "term": {
      "text": "Machine"
    }
  }
}
```

```
GET ml/_search
{
  "query": {
    "term": {
      "text": "machine learning"
    }
  }
}
```

```
GET ml/_search
{
  "query": {
    "term": {
      "text": "Machine Learning"
    }
  }
}
```

```
GET ml/_search
{
  "query": {
    "term": {
      "text": "machine"
    }
  }
}
```



```
GET ml/_search
{
  "query": {
    "term": {
      "text": "machine"
    }
  }
}
```



```
GET ml/_search
{
  "query": {
    "term": {
      "text": "Machine learning"
    }
  }
}
```



```
GET ml/_search
{
  "query": {
    "term": {
      "text": "Machine Learning"
    }
  }
}
```

text field로 색인되었기에 “Machine Learning”은 standard analyzer를 거쳐
“machine”과 “learning”의 토큰나이즈 된다. (“Machine Learning”은 검색 대상이 아니다)
그리고 term query로 검색을 수행하기에 정확히 일치하는 “machine”만 검색이 된다

```
GET ml/_search
{
  "query": {
    "match": {
      "text": "machine"
    }
  }
}
```

```
GET ml/_search
{
  "query": {
    "match": {
      "text": "Machine"
    }
  }
}
```

```
GET ml/_search
{
  "query": {
    "match": {
      "text": "machine learning"
    }
  }
}
```

```
GET ml/_search
{
  "query": {
    "match": {
      "text": "Machine Learning"
    }
  }
}
```

```
GET ml/_search
{
  "query": {
    "match": {
      "text": "machine"
    }
  }
}
```

```
GET ml/_search
{
  "query": {
    "match": {
      "text": "Machine"
    }
  }
}
```

```
GET ml/_search
{
  "query": {
    "match": {
      "text": "machine learning"
    }
  }
}
```

```
GET ml/_search
{
  "query": {
    "match": {
      "text": "Machine Learning"
    }
  }
}
```

text field로 색인되었기에 “Machine Learning”은 standard analyzer를 거쳐
“machine”과 “learning”의 토크나이즈 된다. (“Machine Learning”은 검색 대상이 아니다)
그리고 match query로 검색을 수행하기에 검색어 또한 검색시 standard analyzer를 거친다
그러므로 모든 검색어가 검색이 된다.

특정 접두어로 시작하는 Document 조회

문법

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

예시

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

Wildcard Expression 만족하는 Documents 조회

문법

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

예시

- wildcard (*)

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

- wildcard*?)

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

문법

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

예시

- 기본

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

- 옵션 사용

```
GET /shopping/_search
{
  "query": {
    "fuzzy" : {
      "고객주소_시도" : {
        "value" : "경상북남",
        "fuzziness" : 2
      }
    }
  }
}
```

특정 Numeric Field가 임의의 범위 내에 있는 Documents 조회

문법

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

예시

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

non-null value가 존재하는 Documents 조회

문법

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

예시

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

Lucene Query Syntax를 만족하는 Documents 조회

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

문법

- 기본

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

 **Query String Syntax**

- AND 연산

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

 **Query String Syntax**

Scripted Field가 특정 조건을 만족하는 Document 조회

Management / Kibana

Index Patterns Saved Objects Advanced Settings

+ Create Index Pattern

★ nginx-*



🕒 Time Filter field name: @timestamp

This page lists every field in the **nginx-*** index and the field's associated core type as recorded by Elasticsearch. While this list allows you to view the core type of each field, changing field types must be done using Elasticsearch's [Mapping API](#).

fields (41)

scripted fields (6)

source filters (0)

Filter

All languages ▾

Scripted fields

These scripted fields are computed on the fly from your data. They can be used in visualizations and displayed in your documents, however they can not be searched. You can manage them here and add new ones as you see fit, but be careful, scripts can be tricky!

+ Add Scripted Field

name	lang	script	format	controls
요일_한글	painless	if (doc['@timestamp'].date.dayOfWeek == 1) { return "월" } else if (doc['@timestamp'].date.dayOfWeek == 2) { return "화" } else if (doc['@timestamp'].date.dayOfWeek == 3) { return "수" } else if (doc['@timestamp'].date.dayOfWeek == 4) { return "목" } else if (doc['@timestamp'].date.dayOfWeek == 5) { return "금" } else if (doc['@timestamp'].date.dayOfWeek == 6) { return "토" } else { return "일" }	String	
요일_숫자	painless	doc['@timestamp'].date.dayOfWeek	Number	
시간대	painless	doc['@timestamp'].date.hourOfDay	Number	
access.user_agent	painless	doc['nginx.access.user_agent.name'].value + ' ' + doc['nginx.access.user_agent.major'].value + '.' + doc['nginx.access.user_agent.minor'].value + ' ' + doc['nginx.access.user_agent.patch'].value	String	
요일_local	painless	LocalDateTime.ofInstant(Instant.ofEpochMilli(doc['@timestamp'].value.millis), ZoneId.of('Asia/Seoul')).getDayOfWeek()	String	
시간대_local	painless	LocalDateTime.ofInstant(Instant.ofEpochMilli(doc['@timestamp'].value.millis), ZoneId.of('Asia/Seoul')).getHour()	Number	

Scroll to top

Page Size 25 ▾



위에서 생성했던 Scripted Field를 직접 사용하지는 못한다

Script Query가 특정 조건을 만족하는 Document 조회

문법

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

예시

```
GET /shopping/_search
{
  "query": {
    "script": {
      "script": {
        "source":
          """Instant.ofEpochMilli(doc['주문시간'].value.millis)
            .atZone(ZoneId.of('Asia/Seoul')).hour > 10""",
        "lang": "painless"
      }
    }
  }
}
```

Scripted Field 생성하기 위해 사용했던 코드를 직접 입력해야 한다

여러가지 Query를 복합적으로 사용할 수 있을까?

A : 고객주소_시도 = 서울특별시

Term Query

B : 구매사이트 = 11로 시작

Prefix Query

C : 고객나이 < 30

Range Query

D : 주문시간 > 15

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 참고만 하자

기본 구조는 다음과 같다 (Term Query 예시)

```
GET /shopping/_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

예제3) A AND B를 구해보자

A : 고객주소_시도 = 서울특별시

B : 구매사이트 = 11로 시작

예제4) A AND NOT B를 구해보자

A : 고객주소_시도 = 서울특별시
B : 구매사이트 = 11로 시작

예제5) A OR B를 구해보자

A : 고객주소_시도 = 서울특별시
B : 구매사이트 = 11로 시작

예제6) A AND (B OR C)를 구해보자

- A : 고객주소_시도 = 서울특별시
- B : 구매사이트 = 11로 시작
- C : 고객나이 < 30

예제7) A AND (B OR C OR D 중 2개 이상) 를 구해보자 

A : 고객주소_시도 = 서울특별시

B : 구매사이트 = 11로 시작

C : 고객나이 < 30

D : 주문시간 > 15

아직 Bool Query가 익숙하지 않으면 우선 query string으로 시작하자 

(다만 기능이 제한적이며 복잡해진다)

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

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

꼭 모든 걸 Query DSL로 할 필요는 없다.

Search, Filter, Query DSL을 목적에 맞게 적절히 사용하자

질문 및 Feedback은

gshock94@gmail.com로 주세요