Elastic Stack을 활용한 Data Dashboard 만들기

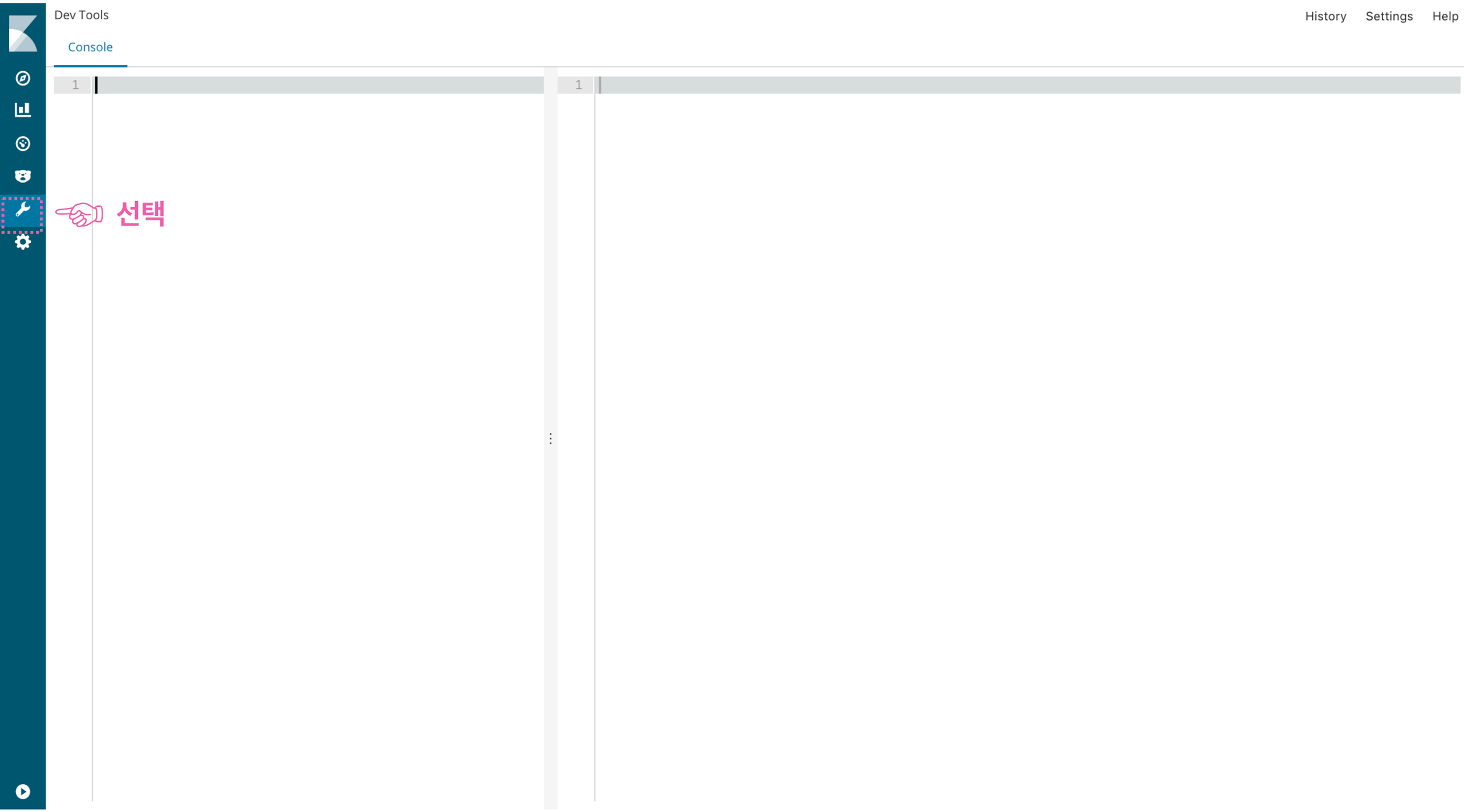
Week 4 - Elasticsearch API를 활용해보자

Fast Campus

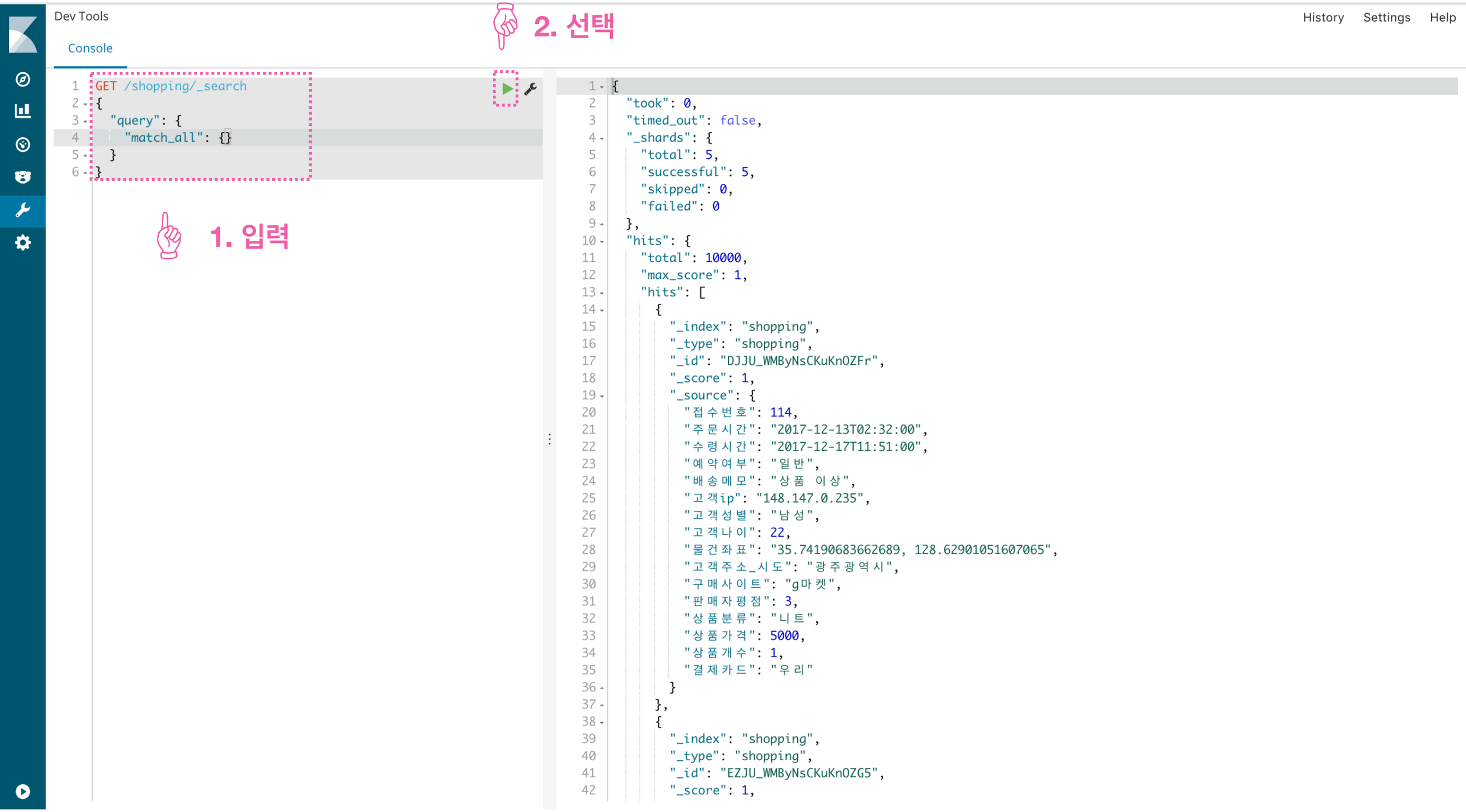
Kibana Dev Tools를 (간단히) 살펴보자

오늘 배우는 API는 (대부분) 여기에 작성한다

Kibana에 접속해서 Dev Tools 화면으로 가자

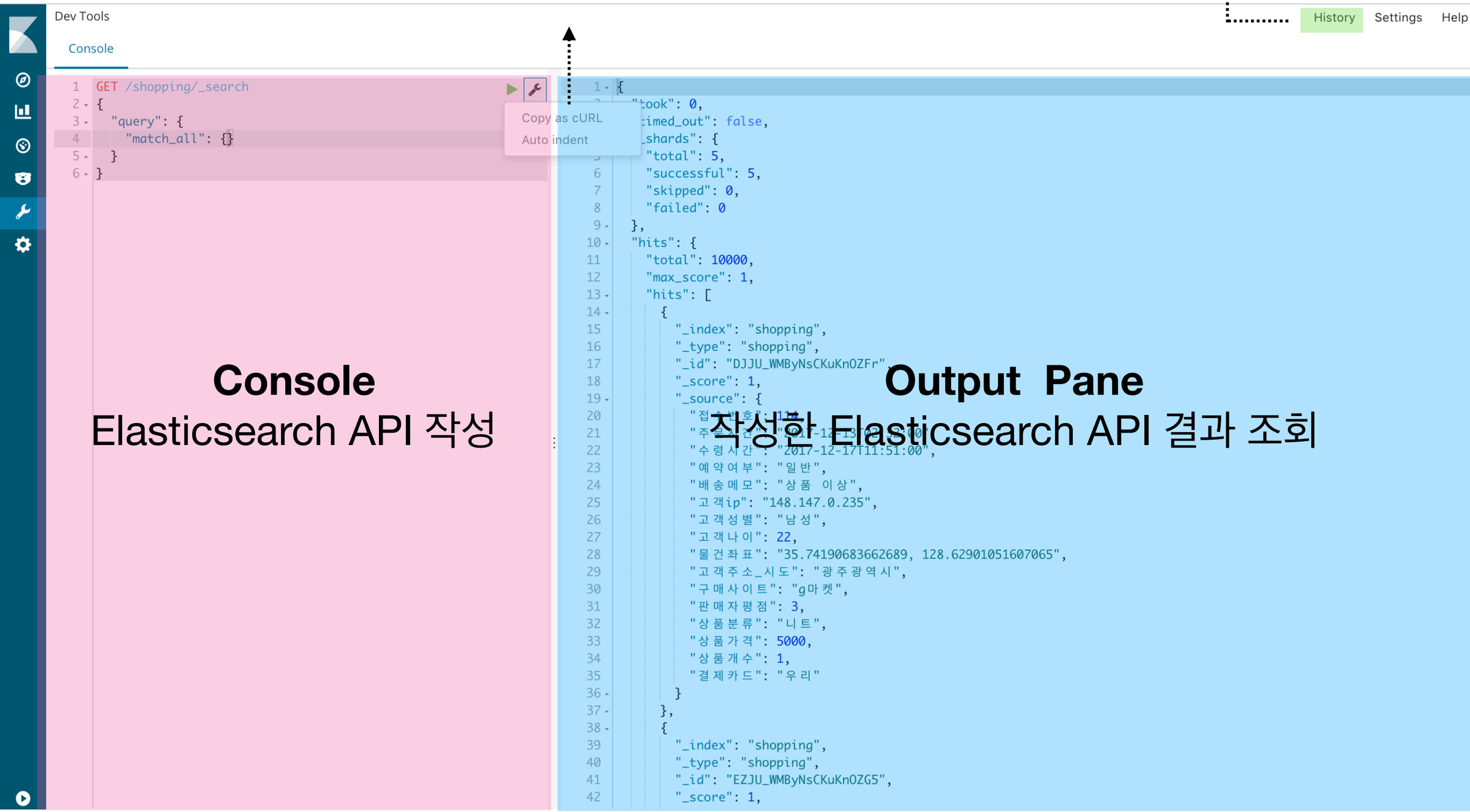


(우선) 다음과 같이 입력하고 녹색 버튼을 눌러보자 👑



과거에 작성한 API 이력 조회

cURL 명령어로 복사



Console
Elasticsearch API 작성

Output Pane
작성한 Elasticsearch API 결과 조회

Dashboard를 구축/운영 함에 있어

알면 좋은 (최소한의) datatype을 살펴보자

-10^{50}

5^{-1}

17^{17}

1

100 0.01

2^{-500}

가방 // 에 // 들어가 // 시 // ㄴ다

직장인

서울

나/는/밥/을/먹/는다

패스트캠퍼스

하늘/을/나/는/자동차

엘라스틱서치

[127.02, 37.51]

{"lat" : 37.51, "lon" : 127.02}

"37.51, 127.02"

"2017-11-28T13:50:333Z"

"2017-11-11"

1420070420134

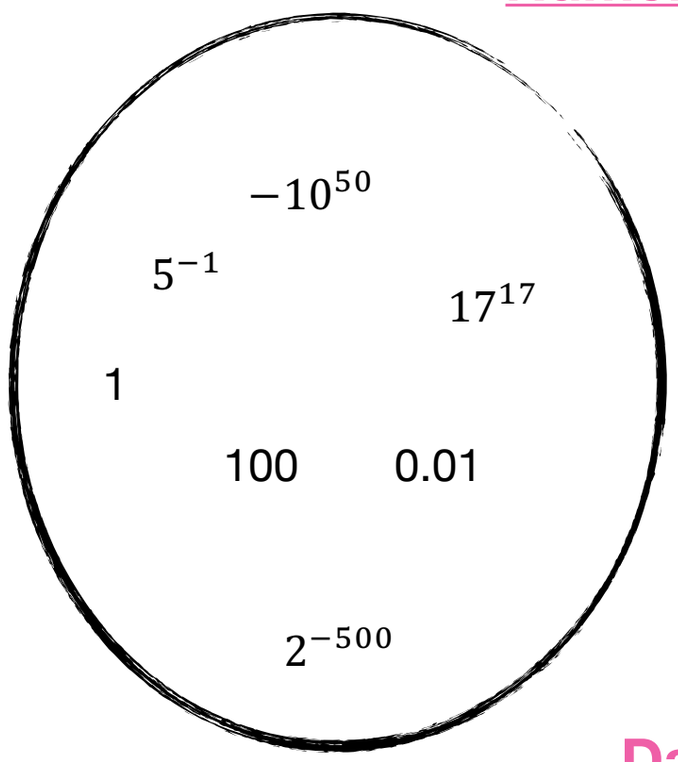
29.175.52.121

42.3.19.175

130.152.206.29

95.100.125.95

Numeric



String

가방 // 에 // 들어가 // 시 // ㄴ다

직장인

서울

나/는/밥/을/먹/는다

패스트캠퍼스

하늘/을/나/는/자동차

엘라스틱서치

Geo-point

[127.02, 37.51]

{"lat" : 37.51, "lon" : 127.02}

"37.51, 127.02"

Date

"2017-11-28T13:50:333Z"

"2017-11-11"

1420070420134

IP

29.175.52.121

42.3.19.175

130.152.206.29

95.100.125.95

이 때 주의할 Type은 Numeric과 String

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

Numeric datatypes는 크게 정수형과 부동 소수점형으로 나뉜다 👑

Numeric datatypes

edit

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

어떤 Type을 사용해야 될까?

Numeric datatypes

edit

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 $-32,768$ 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`.
`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, 넉넉한 (=안전하게) type을 사용하자

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

가지고 있는 Numeric Data의 성격을 잘 안다면, 최적화에 집중하자 **심화**

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)

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

심화

Dev Tools

Console

```
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 : 실제 값을 표현하기 부족한 Precision을 선택한 경우 👑 **심화**

	“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

부동소수점 : 실제 값을 표현하기 부족한 Precision을 선택한 경우 **심화**

“test-double” field			“test-half-float” field		
입력값	3.111111222: test-double	3.111111: test-double	3.1: test-double	3.111111222: test-half-float	3.111111: test-half-float	3.1: test-half-float
인식값	**3.111111222**	**3.111111**	**3.1**	**3.111328125**	**3.111328125**	**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

"패스트캠퍼스 엘라스틱서치" => text field와 keyword field 비교 👑 **심화**

	“test-text” field	**“test-keyword” field**
data type	**text**	**keyword**
indexing	“패스트캠퍼스 엘라스틱서치”	

search
(=match query)
	`GET /my_index/_search`
	`{`
	`  "query": {`
	`  "term": {`
	`  "test-text": "패스트캠퍼스"`
	`  }`
	`  }`
	`}`
		`GET /my_index/_search`
		`{`
		`  "query": {`
		`  "term": {`
		`  "test-keyword": "패스트캠퍼스"`
		`  }`
		`  }`
		`}`

result	**O**	**X**

"패스트캠퍼스 엘라스틱서치" => text field와 keyword field 비교 👑 **심화**

	“test-text” field	**“test-keyword” field**
data type	**text**	**keyword**
indexing	“패스트캠퍼스 엘라스틱서치”	

search
(=match query)
	`GET /my_index/_search`
	`{`
	`  "query": {`
	`  "term": {`
	`  "test-text": "패스트캠퍼스"`
	`  }`
	`  }`
	`}`
		`GET /my_index/_search`
		`{`
		`  "query": {`
		`  "term": {`
		`  "test-keyword": "패스트캠퍼스"`
		`  }`
		`  }`
		`}`

result	**O**	**X**

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

심화

~~API 학습 후에~~

조금 특별한 Data Type도 살펴보자

(= Complex datatypes 👑)

심화

데이터를 계층적으로 저장할 수 없을까? 👑

object datatype에 데이터를 색인해보자 👑 심화

```
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

명시적으로 mapping을 설정할 필요는 없지만 아래와 같이 할 수도 있다 👑

심화

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

심화

데이터를 배열 형태로 저장할 수 없을까? 👑

array datatype에 데이터를 색인해보자 👑 심화

```
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"]
}
```

명시적으로 mapping을 설정할 필요는 없지만 아래와 같이 할 수도 있다 👑

심화

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

심화

데이터를 (앞과 살짝 다른) 배열 형태로 저장할 수 없을까 ? 👑

단, 이번에는 명시적으로 mapping을 생성해야 한다 👑

심화

```
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 type과 달리,

상호 독립적으로 색인/검색될 수 있다!

실제 예시를 통해 비교해보자

1. mapping 심화

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" }
          }
        }
      }
    }
  }
}
```

2. indexing 심화

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

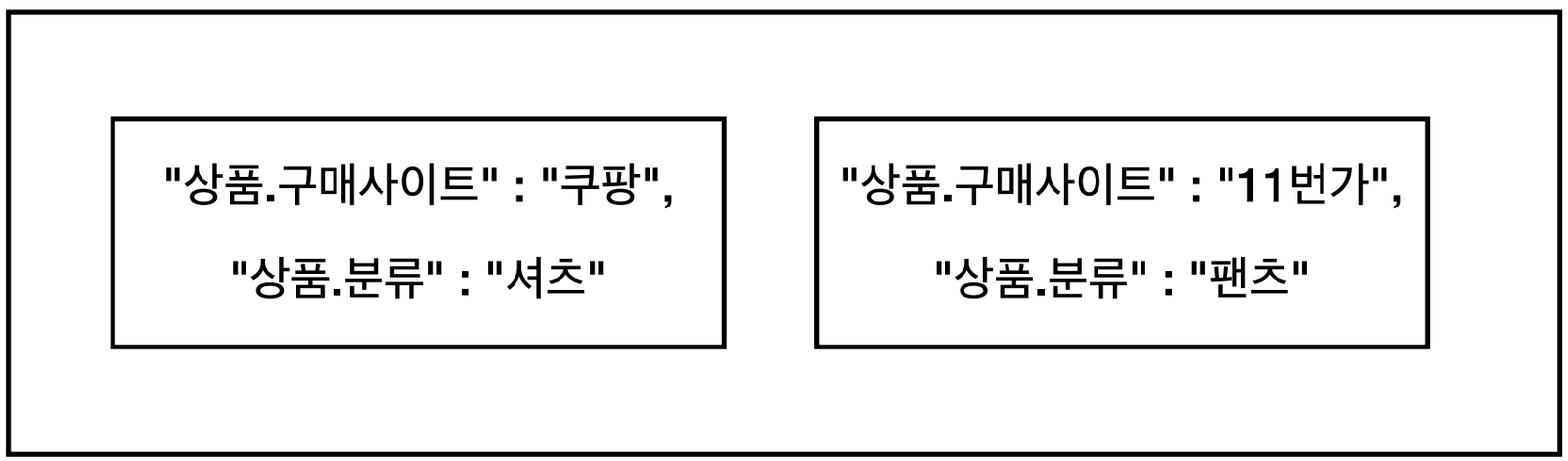
```
GET array/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "상품.구매사이트": "11번가"
          }
        },
        {
          "match": {
            "상품.분류": "셔츠"
          }
        }
      ]
    }
  }
}
```

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

왜 검색이 되지?

3.1 array datatype 심화



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

"상품.구매사이트" : "쿠팡",
"상품.분류" : "셔츠"

"상품.구매사이트" : "11번가",
"상품.분류" : "팬츠"

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

3.2 nested datatype 👑 심화

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

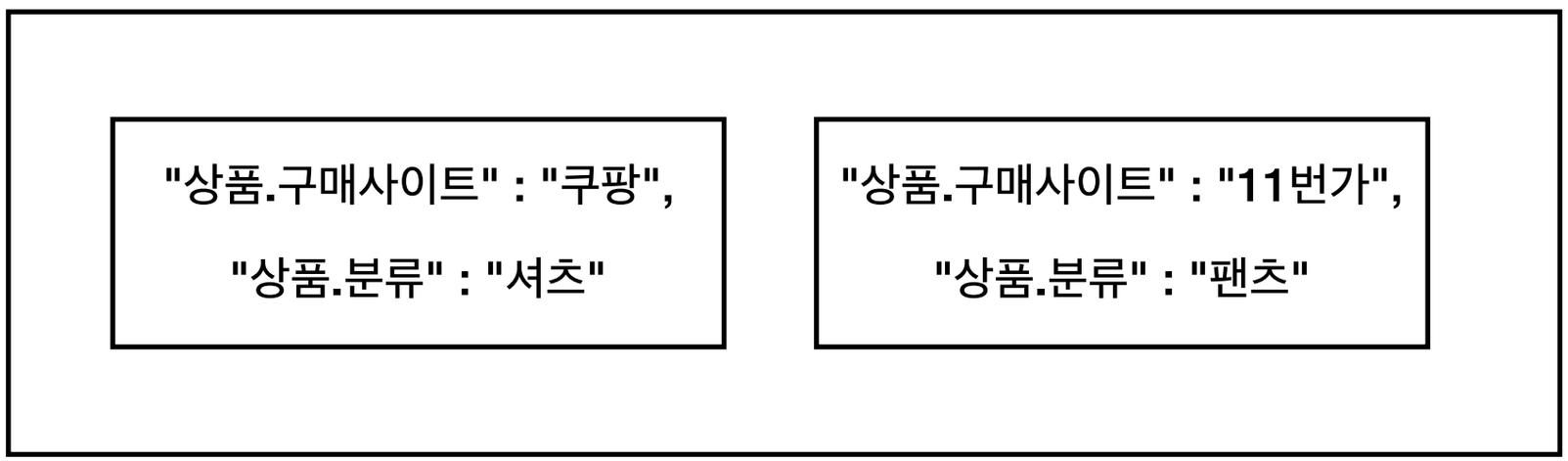
3.2 nested datatype 👑 심화

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

3.2 nested datatype 심화



원래 데이터는 위와 같은 두 object를 가진 nested 형태였다.

3.2 nested datatype **심화**

"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

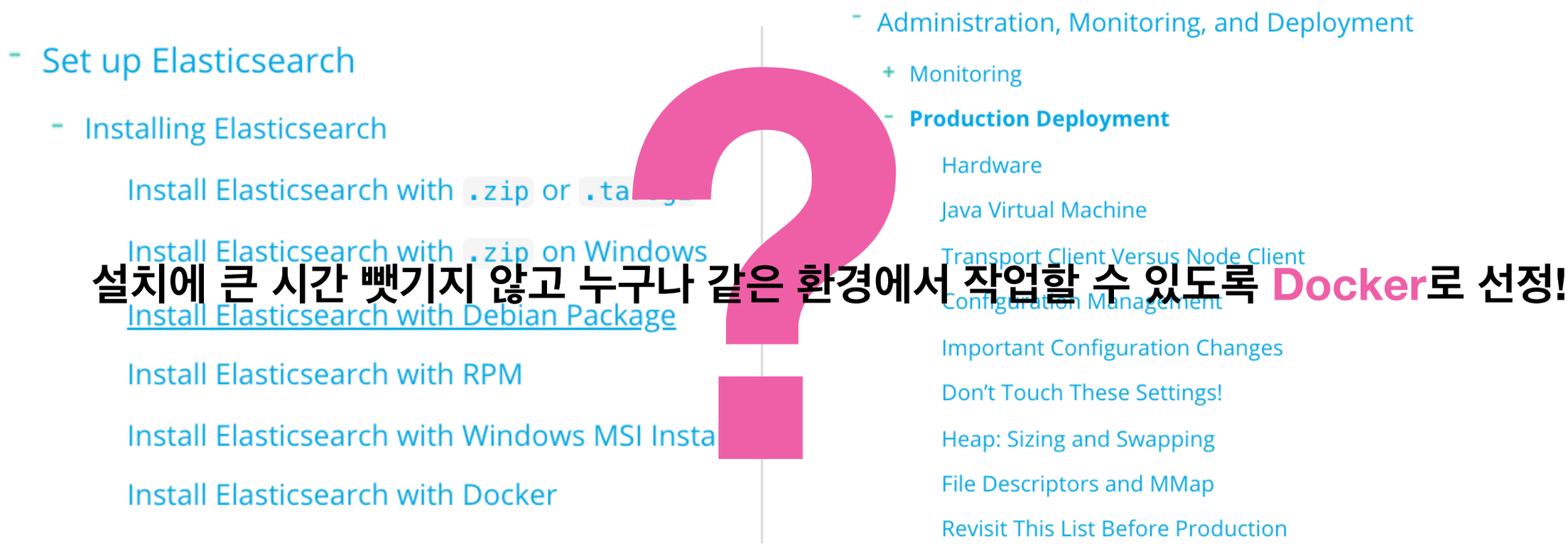
∩
- 상품.구매사이트 = 쿠팡
- 상품.분류 = 셔츠

Array datatype과 Nested datatype은

위 예시를 참고해서 목적에 맞게 사용하자

Elastic Stack을 직접 설치/운영해보자

어떤 방법으로 할까? 👑



설치에 큰 시간 뺏기지 않고 누구나 같은 환경에서 작업할 수 있도록 Docker로 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.

우선 AWS EC2 Instance에 접속하자 👑

- Mac OS : Terminal
- Windows : Putty

Elastic Stack을 설치해보자 👑

다양한 Elasticsearch API 중에서

Indices, Document, Search API를 알아보자

Indices API로 무얼 할 수 있을까?

- **Index 생성/삭제**
- **Mapping 설정**

Dev Tools 페이지로 이동해서 하나씩 입력하고 결과를 확인하자

Index 생성 👑

문법

```
PUT {Index 이름}
```

예시

```
PUT week4_higee
```

Index 삭제 👑

문법

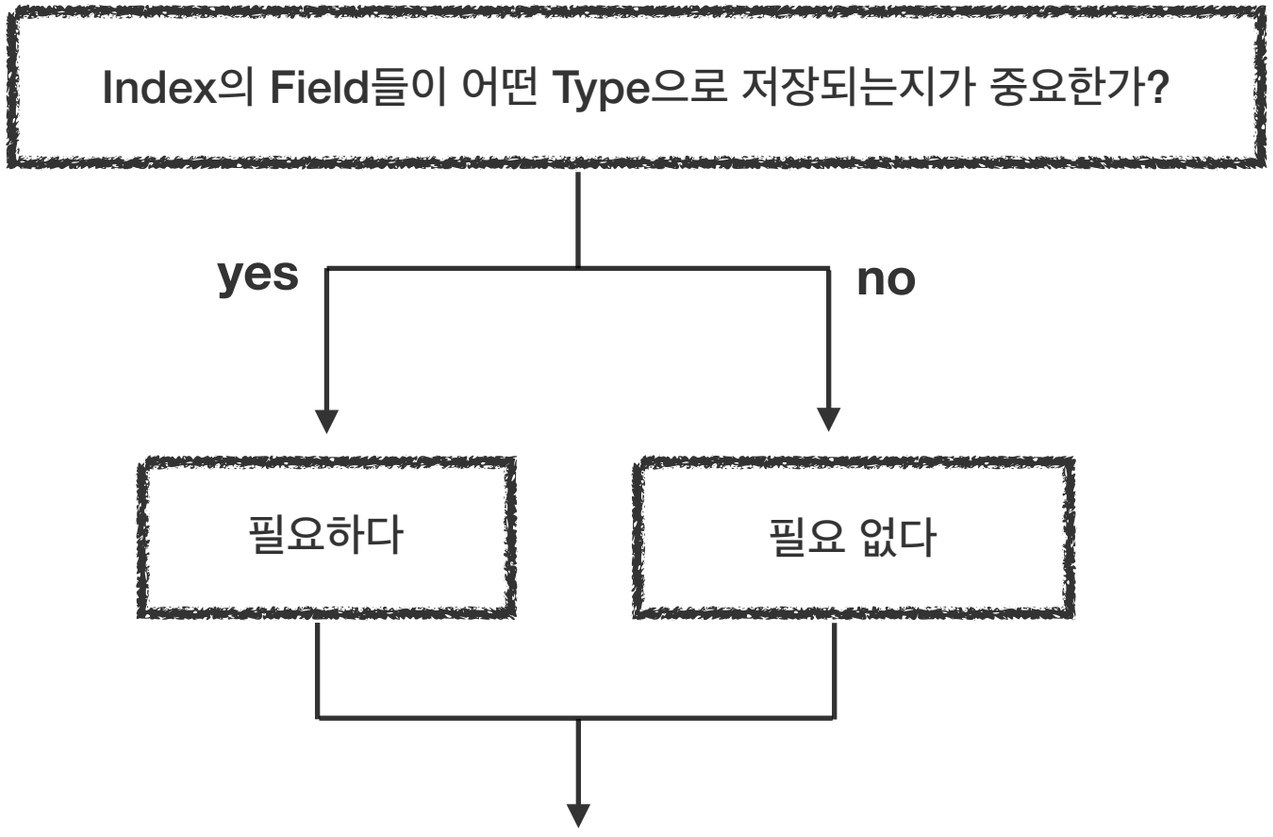
```
DELETE {Index 이름}
```

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 부여**한다
  - 단, 한 번 설정된 **Data Type은 변경이 불가능**하다
  - 단, 데이터가 색인된 후에도 **새로운 Field에 대한 Mapping은 추가**할 수 있다
- 그러므로 일반적으로는 **Index 생성하는 시점**에 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"
    }
  }
}
```

Template 활용해서 자동으로 Mapping 추가 👑 심화

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"
        }
      }
    }
  }
}
```

심화

Template 생성 ≠ Index 생성

≠ Mapping 생성

Template에서 정의한 Index Pattern에 해당하는 **Index가 생성될 때,**

(Template을 활용해서 사전 정의한) **Mapping이 적용된다**

그렇다면 Template은 언제 유용할까?

심화

비슷한 이름의 Index가 **정기적**으로 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 Index 마다 직접 Mapping을 추가해야 한다

Mapping 확인 👑

문법

```
GET {Index 이름}/_mapping
```

예시

```
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`

예제1

1. 실습 서버의 shopping index의 mapping을 확인하고,
2. 그와 같은 mapping을 갖는 index를 자기 서버에 생성하자
3. 완료한 경우, 카카오톡방에 ip주소와 함께 알려주세요 (**데이터 전송 목적**)
4. dashboard 👑 와 visualization 👑 import
5. 아래와 같은 scripted field를 생성하자
   - 연령대
     - 10대 : 고객나이 < 20
     - 20~30대 : 20 ≤ 고객나이 < 40
     - 40대 이상 : 고객나이 ≥ 40
   - 배송소요시간 : (수령시간 - 주문시간)의 값을 시간(hour)로 환산환 단위
   - 주문시간_시간대 : 주문시간 값을 기준으로 한국 시간대를 추출
   - 주문시간_요일 : 주문시간 값을 기준으로 한국 요일을 추출

Document API로 무얼 할 수 있을까?

Document API로 무얼 할 수 있을까?

- **Document 생성 Ⓒ**
- **Document 조회 Ⓡ**
- **Document 수정 Ⓤ**
- **Document 삭제 Ⓓ**
- **Document 복사**

Document 추가 (지정 ID) 👑

문법

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

예시

```
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

```
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 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

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

예시

```
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 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"
  }
}
```

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

Request Body Search에서 사용되는 Domain Specific Language

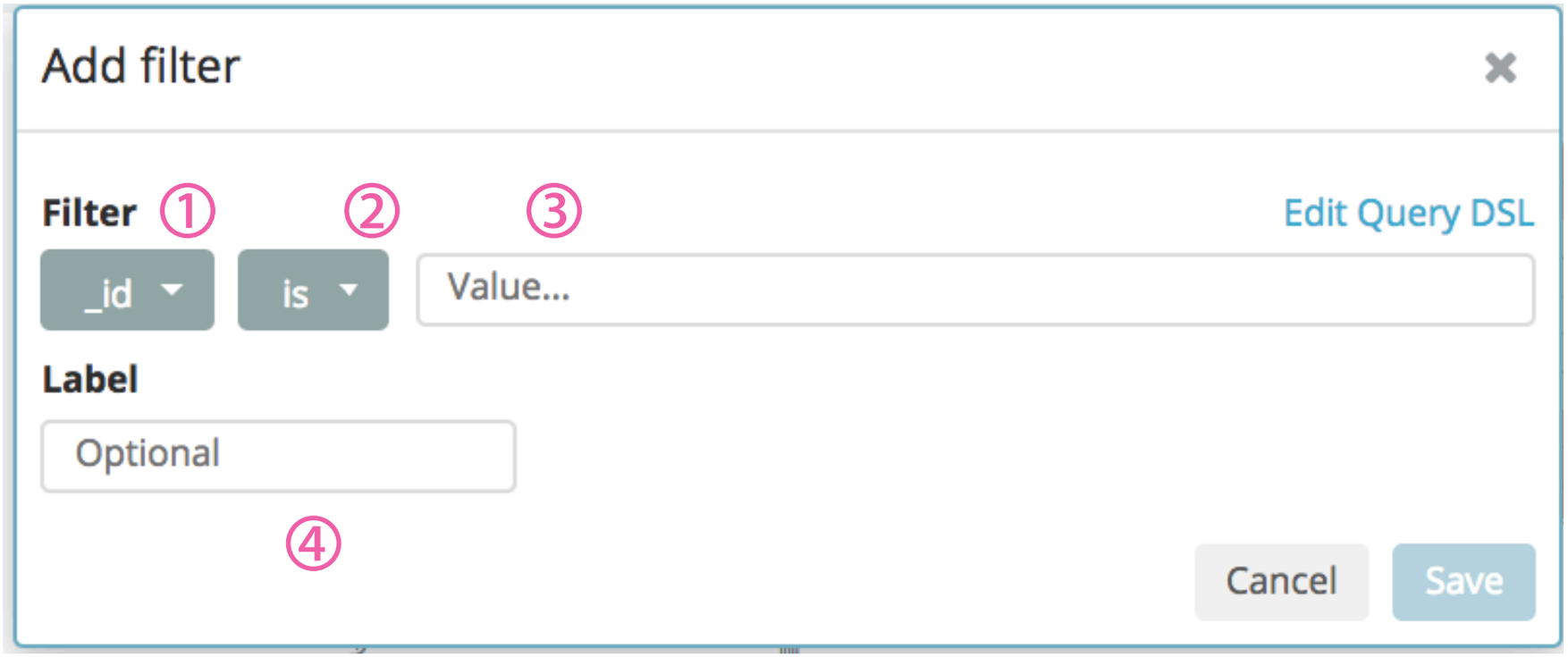
Match All Query	Full Text Queries	Term Level Queries	Specialized Queries	Compound Queries
match-all	match query-string ⋮	exists fuzzy prefix range term terms wildcard ⋮	script ⋮	bool ⋮

간략히나마 뭘 위한건지는 알겠는데

Dashboard를 구축/운영하는데 왜 필요하지?

Filter 기능 강화를 위해서

Filter를 다시 보자



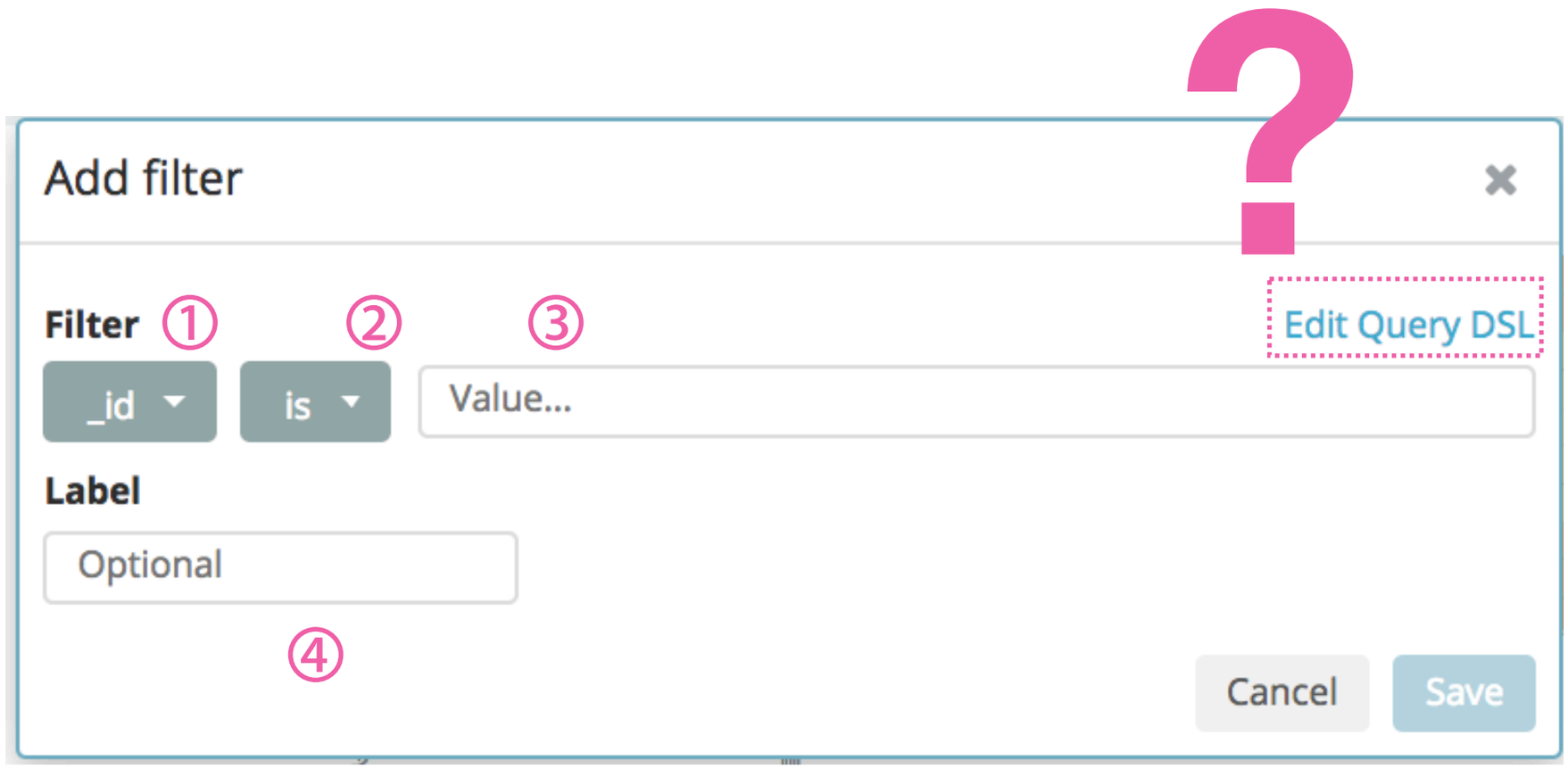
① Filter 적용할 Field 선택

② 적용할 Operator 선택 (다음 페이지 참조)

③ Filter에 적용하려는 Value 입력

④ (여러 Filter 구분하기 위한) 이름 입력

Filter를 다시 보자



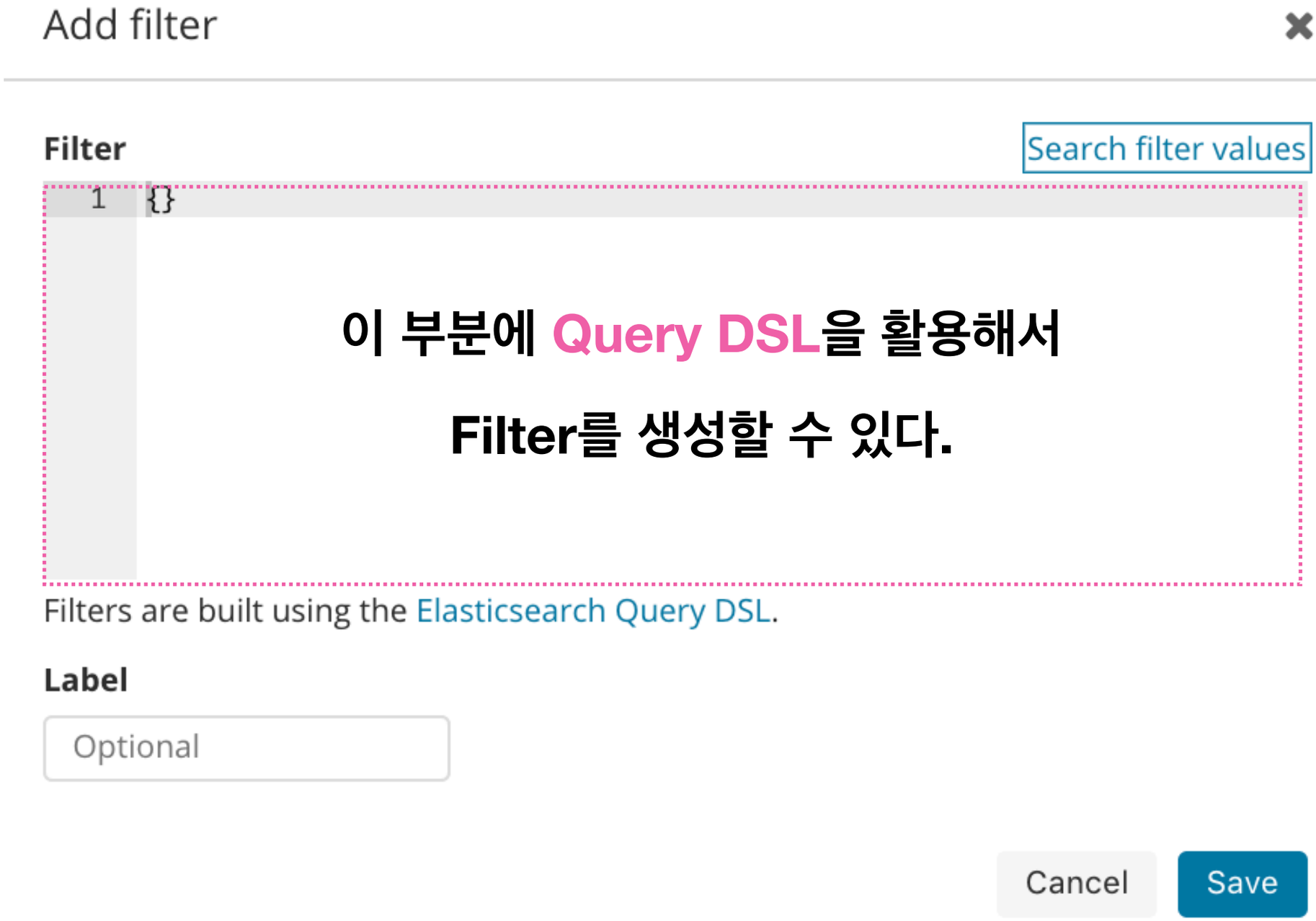
① Filter 적용할 Field 선택

② 적용할 Operator 선택 (다음 페이지 참조)

③ Filter에 적용하려는 Value 입력

④ (여러 Filter 구분하기 위한) 이름 입력

Edit Query DSL



예를 들어, 아래와 같은 Query DSL을 작성하면

Add filter

Filter

Search filter values

```
{
  "query" : {
    "term" : {
      "nginx.access.response_code" : "200"
    }
  }
}
```

1. 입력

Filters are built using the Elasticsearch Query DSL.

Label

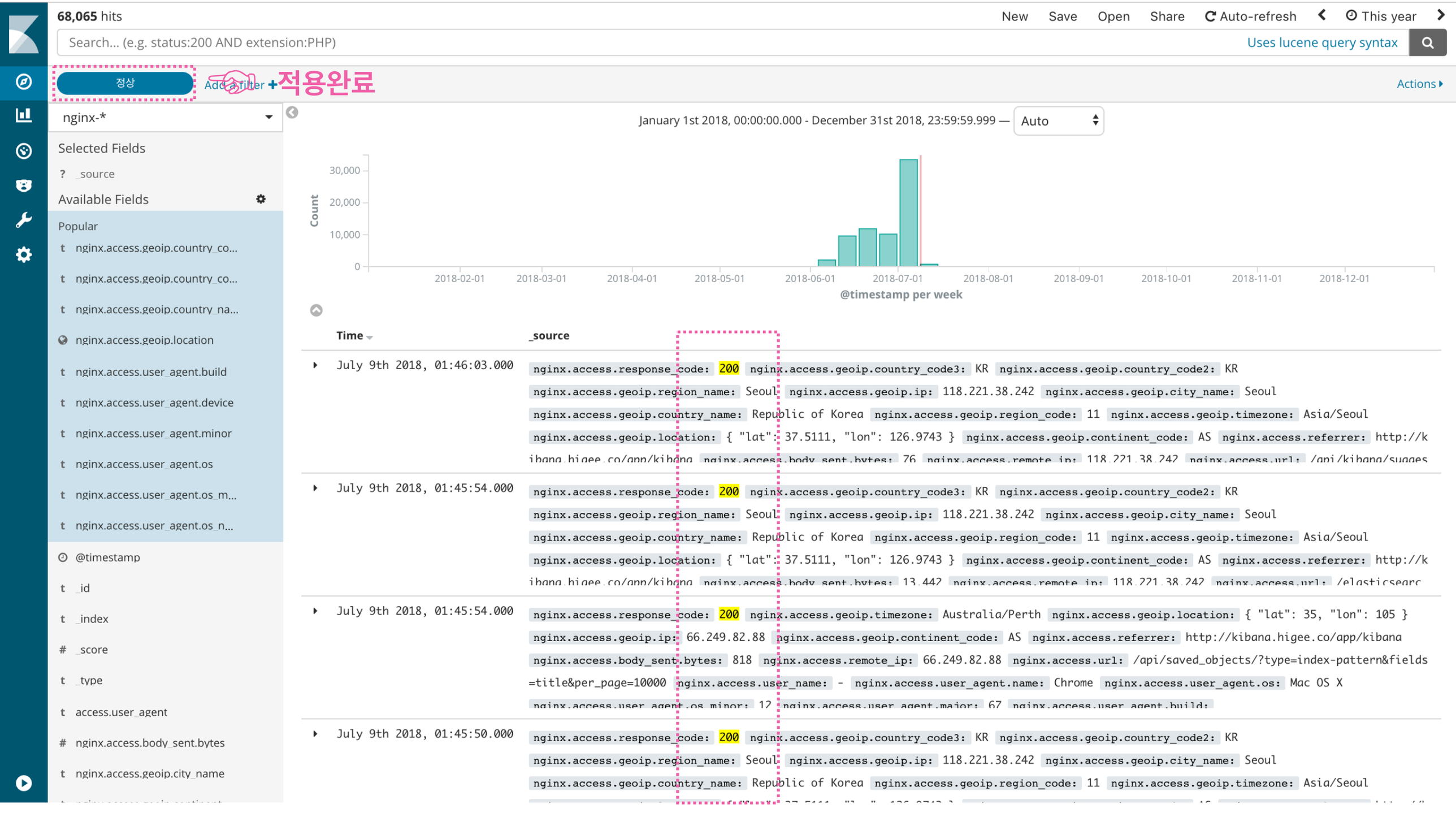
정상

2. 입력

Cancel Save

3. 선택

이런 결과가 나온다



데이터 확인

즉, Query DSL을 이용하면

!

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		✓	✓

AND 연산

OR 연산

Scripted Field

Wildcard 검색

Fuzzy/Proximity 검색

모든 Documents 조회 👑

문법

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

예시

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

```
{
  "took": 0,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "skipped": 0,
    "failed": 0
  },
  "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

모든 Documents 조회 👑

문법

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

예시

```
GET /shopping/_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" : {}
  }
}
```

Documents는 어떤 순서로 정렬되지? 👑 **심화**

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"
      }
    }
  ]
}
```

왜 Documents가 10개 밖에 표시가 안되지? 👑 **심화**

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

특정 Field의 Value만 보고 싶다면? 👑 **심화**

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"
      ]
    }
  }
}
```

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"
    }
  }
}
```

검색어와 유사한 Documents 조회 👑

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"
    }
  }
}
```

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

“Machine Learning”

“machine”

“Learning”

"Machine”

“machine learning”

“learning”

“Machine Learning”이라는 데이터를 색인하고 아래와 같은 검색어로 검색해보자 심화

검색어

어떻게 될까?

“machine” “Learning”

"Machine”

“machine learning” “learning”

결과는 아래 설정을 어떻게 하냐에 따라 달라진다 **심화**

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
	Machine	x
	machine learning	x
	Machine Learning	**o**
match	machine	x
	Machine	x
	machine learning	x
	Machine Learning	**o**
term	machine	**o**
	Machine	x
	machine learning	x
	Machine Learning	x
match	machine	**o**
	Machine	**o**
	machine learning	**o**
	Machine Learning	**o**

먼저 매핑을 만들고 색인하자 **심화**

- 매핑 👑

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

- 색인 👑

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

keyword field에 대해 term query로 검색해보자 👑 **심화**

```
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 대해 term query로 검색해보자 👑 **심화**

```
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

keyword field에 대해 match query로 검색해보자 👑 **심화**

```
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"
    }
  }
}
```

keyword field에 대해 match query로 검색해보자 👑 **심화**

```
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 field에 대한 검색이기에 정확히 일치하는 것만 검색된다.
즉 , 검색 수행 시 검색어에 대한 analyze를 거치지 않는다.

text field에 대해 term query로 검색해보자 👑

심화

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

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

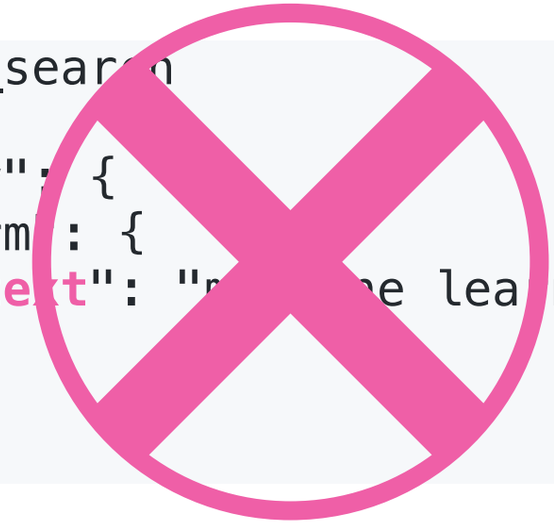
```
GET ml/_search
{
  "query": {
    “term": {
      "text": "machine learning”
    }
  }
}
```

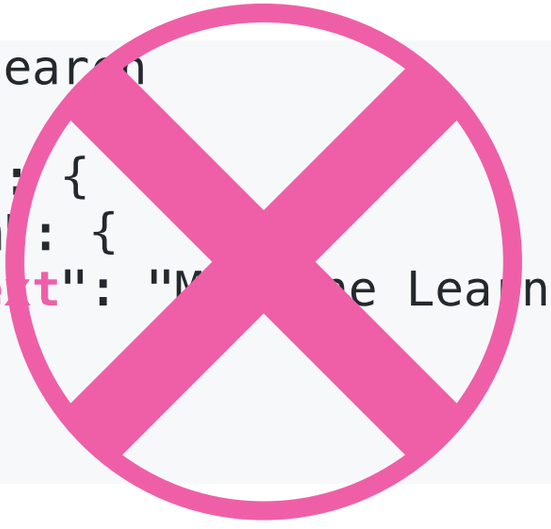
```
GET ml/_search
{
  "query": {
    “term": {
      "text": "Machine Learning”
    }
  }
}
```

text field에 대해 term query로 검색해보자 👑

심화

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





text field로 색인되었기에 “Machine Learning”은 standard analyzer를 거쳐

“machine”과 “learning”의 토크나이즈 된다. (“Machine Learning”은 검색 대상이 아니다)

그리고 term query로 검색을 수행하기에 정확히 일치하는 “machine”만 검색이 된다

text field에 대해 match query로 검색해보자 👑 **심화**

```
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 field에 대해 match query로 검색해보자 👑 **심화**

```
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.

Lucene Query Syntax를 만족하는 Documents 조회 👑

문법

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

예시

- 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-*

★ 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 Occurence Type을 알아보자 👑

Bool Query Occurence	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**

1. A AND B를 구해보자 👑

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

```
GET /shopping/_search
{
  "query": {
    "bool": {
      "must": [
        { "term": { "고객주소_시도": “서울특별시"}},
        { "prefix": { "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": [
        { "prefix": { "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" }},
        { "prefix": { "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": [
        { "prefix": { "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 : 주문시간 > 15

```
GET /shopping/_search
{
  "query": {
    "bool": {
      "must": [
        {"term": {"고객주소_시도": "서울특별시"}}
      ],
      "should": [
        {"prefix": {"구매사이트" : "11"}},
        {"range": {"고객나이": { "lt": 30}}},
        {"script": {
           "script" : {
             "source" : "Instant.ofEpochMilli(doc['주문시간'].value.millis).atZone(ZoneId.of('Asia/Seoul')).hour>15"}}}
      ],
      "minimum_should_match": 2
    }
  }
}
```

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”,
      "analyze_wildcard": true
    }
  }
}
```

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

꼭 모든 걸 Query DSL로 할 필요는 없다.

Search, Filter, Query DSL을 목적에 맞게 적절히 사용하자

질문 및 Feedback은

gshock94@gmail.com로 주세요