# Implementing Queries

**Dan Geabunea**
SENIOR SOFTWARE DEVELOPER

@romaniancoder    www.romaniancoder.com

## Overview

Understand the query object

How to create and execute queries

How to implement full text search

Demo: Implementing Mongo queries in Spring applications

## Query Execution

| Without indexes | With indexes |
|---|---|
| Collection scan, each document is evaluated | Does not scan every document in collection |
| Slow searches | Fast searches |
| Fast inserts, updates | Slower inserts/updates |

## Mongo Query Object

The query object is the most powerful and flexible component for Mongo CRUD operations

# Query Components

Criteria for filtering data

Sorting definition for ordering data

Paging definition for splitting data

## Mongo Query Definition Example

```
Query byAge = Query
     .query(Criteria.where(“age").gt(18))
     .with(Sort.by(Sort.Direction.DESC, “age"))
     .with(PageRequest.of(1, 10));
```

```
Criteria.where("age").gt(18)
```

- Left operand (data field)
- Operator(s)
- Right operand (filter value)

## Mongo Filter Operators

| | | |
|---|---|---|
| is / ne<br>(equality) | lt / lte<br>(less than) | gt / gte<br>(greater than) |
| in<br>(value in list) | exists<br>(has value) | regex<br>(patterns) |

MongoTemplate is the component that executes requests to a Mongo database

## MongoTemplate

A simple injectable class following the standard template pattern in Spring

It allows us to perform CRUD operations on data

It allows us to execute commands against a Mongo database

It's very powerful, but also low level.

## How to Create and Execute Queries

## Mongo Query Execution Example

```
Query byAge = Query
    .query(Criteria.where(“age").gt(18))
    .with(Sort.by(Sort.Direction.DESC, “age"))
    .with(PageRequest.of(1, 10));

List<Person> people = this.mongoTemplate.find(byAge, Person.class);
```

## Fetching Data with Mongo Template

1

The query definition with filters, sorts, etc.

2

Decide the outcome of the query

3

Class type that contains Mongo annotations

```
MongoTemplate template = …

Query q = …


template.findAll(Person.class)


template.find(q,Person.class)


template.findOne(q,Person.class)


template.count(q,Person.class)
```

◀ Define a query, inject the template

◀ Fetch all persons

◀ Fetch all persons matching query

◀ Fetch a single person matching query

◀ Count all people matching query

## Aircraft Class

```
@Document class Aircraft {
    @Id private String id;
    @Indexed(unique=true) private String code;
    private String family;
    private int nbSeats;
    private Engine engine;
}

class Engine { private boolean needsMaintenance;}
```

## Find All Documents

```
List<Aircraft> res = mongoTemplate.findAll(Aircraft.class);

// or

Query allAircraft = new Query();
List<Aircraft> res = mongoTemplate.find(allAircraft, Aircraft.class);
```

## Find Single Document

```
Aircraft a = mongoTemplate.findById(id, Aircraft.class);

// or

Query byCode = Query.query(
    Criteria.where("code").is("a234b")
);
Aircraft a = mongoTemplate.findOne(byCode, Aircraft.class);
```

## Filter with Multiple Operators

```
Query nbSeatsBetween = Query.query(Criteria.where("nbSeats")
                        .gt(100)
                        .lt(300)
);

List<Aircraft> res = mongoTemplate.find(nbSeatsBetween, Aircraft.class);
```

## Multiple Criteria: Or

```
Query bigAircraftOr737 = Query.query(new Criteria()
    .orOperator(Criteria.where("family").is('737'),
                Criteria.where("nbSeats").gte(250))
);

List<Aircraft> res = mongoTemplate.find(bigAircraftOr737, Aircraft.class);
```

## Multiple Criteria: And

```
Query bigAircraftAnd737 = Query.query(new Criteria()
    .andOperator(Criteria.where(“family").is(‘737’),
                 Criteria.where(“nbSeats").gte(250))
);

List<Aircraft> res = mongoTemplate.find(bigAircraftAnd737,Aircraft.class);
```

## Filter Subdocuments

```
// engine is a property of Aircraft
// needsMaintenance is a property of Engine
Query engineNeedingMaintenance = Query.query(
    Criteria.where(“engine.needsMaintenance").is(true)
);

List<Aircraft> res = mongoTemplate.find(
                            engineNeedingMaintenance,
                            Aircraft.class);
```

## Sorting the Results

```java
Query moreThan100Seats = Query
                 .query(Criteria.where("nbSeats").gt(100))
                 .with(Sort.by(Sort.Direction.ASC, "nbSeats"));

List<Aircraft> res = mongoTemplate.find(moreThan100Seats, Aircraft.class);
```

# Paging the Results

```
Query moreThan100Seats = Query
                .query(Criteria.where("nbSeats").gt(100))
                .with(Sort.by(Sort.Direction.ASC, "nbSeats"))
                .with(PageRequest.of(1,20));

List<Aircraft> res = mongoTemplate.find(moreThan100Seats, Aircraft.class);
```

## Implementing Full Text Search

## Text Indexes

Work on properties of type string or arrays of strings

You can text index properties across the whole object graph

Pay attention to weights as they may change the order or relevance of a found document

## It Begins with Annotations

```
@Document
class Profile{
    @Id private String id;
    private String name;
    @TextIndexed private String title;
    @TextIndexed private String aboutMe;
}
```

```
{name:”Dan”, title: “Java Developer”, aboutMe: “I am a programmer”},

{name:”Java Guru”, title: “C# Developer”, aboutMe: “I am a programmer”},

{name:”John”, title: “Developer”, aboutMe: “I am a Java programmer”}
```

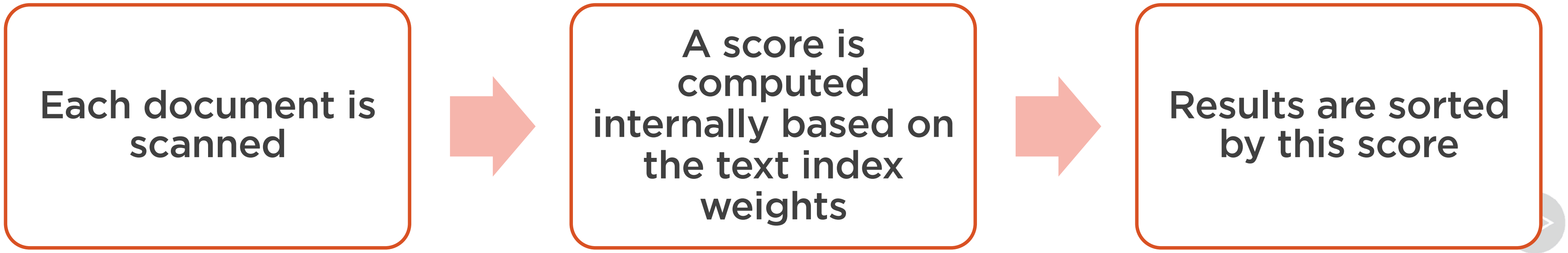
## Let’s Search by ‘Java’

**Output:**

- Dan, John
- John, Dan

```
{name:”Dan”, title: “Java Developer”, aboutMe: “I am a programmer”},
{name:”Java Guru”, title: “C# Developer”, aboutMe: “I am a programmer”},
{name:”John”, title: “Developer”, aboutMe: “I am a Java programmer”}
```

## How Does It Work?


Each document is scanned
A score is computed internally based on the text index weights
Results are sorted by this score

## Equal Weights

```
@Document
class Profile{
    @Id private String id;
    private String name;
    @TextIndexed private String title;
    @TextIndexed private String aboutMe;
}
```

## Text Index Weighting

```
@Document
class Profile{
    @Id private String id;
    private String name;
    @TextIndexed private String title;
    @TextIndexed(weight=2) private String aboutMe;
}
```

```
{name:”Dan”, title: “Java Developer”, aboutMe: “I am a programmer”},

{name:”Java Guru”, title: “C# Developer”, aboutMe: “I am a programmer”},

{name:”John”, title: “Developer”, aboutMe: “I am a Java programmer”}
```

## Let’s Search by ‘Java’

**Output:**

- John
- Dan

## Executing a Full Text Search

```
TextCriteria textCriteria = TextCriteria
                .forDefaultLanguage()
                .matching(text);

Query byFreeText = TextQuery.queryText(textCriteria)
                .sortByScore() // internal score computed by Mongo
                .with(PageRequest.of(0, 3));

return mongoTemplate.find(byFreeText, FlightInformation.class);
```

Demo

**Implementing and executing Mongo queries in Spring applications:**

- Filtering
- Sorting
- Paging
- Full text search

## Summary

**Define queries**

- Query
- Criteria / Sorting / Paging

**Execute queries**

- MongoTemplate

**Implement full text search**

- Using @TextIndex
- How weighting works

“Fetching data is good.
Creating, updating and removing data is better.”

**Me**