**Spring Data**

Repository defined as interfaces.

1. MongoTemplate:

Predefined methods.

Query q= new Query(): to return all documents.

**Upsert:** insert+update (if no result found, will do insert, else update)

**Remove documents:** remove

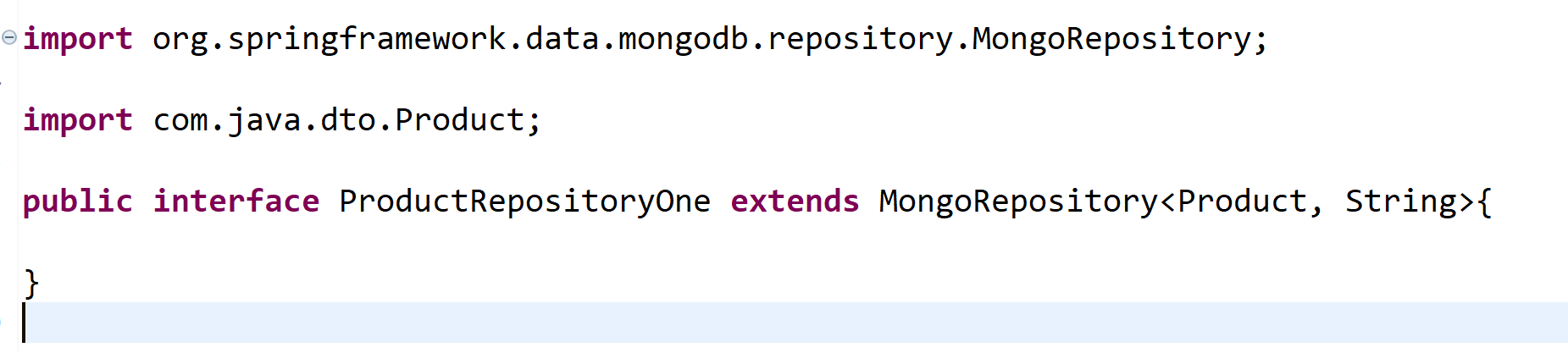
Retrieve doc: findById, findOne, findAll,

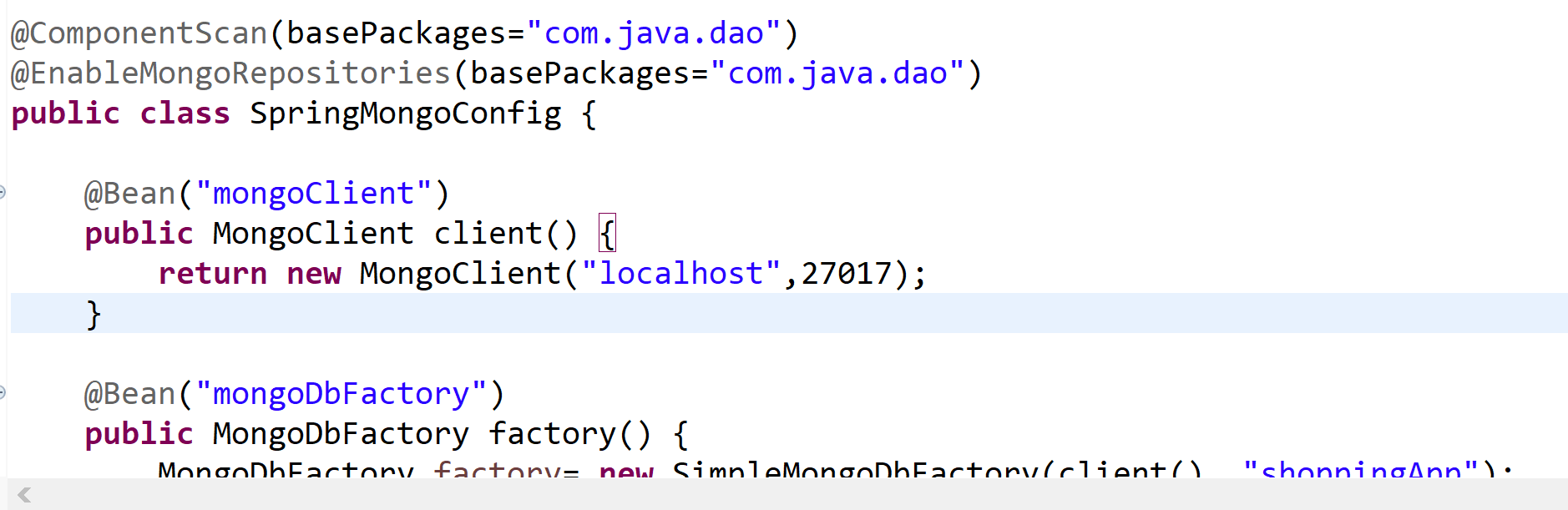
Mongoimport –db dbname –collection name –username u1 –password p1 –drop –file file1.txt

//to drop doc which already exist within that collection

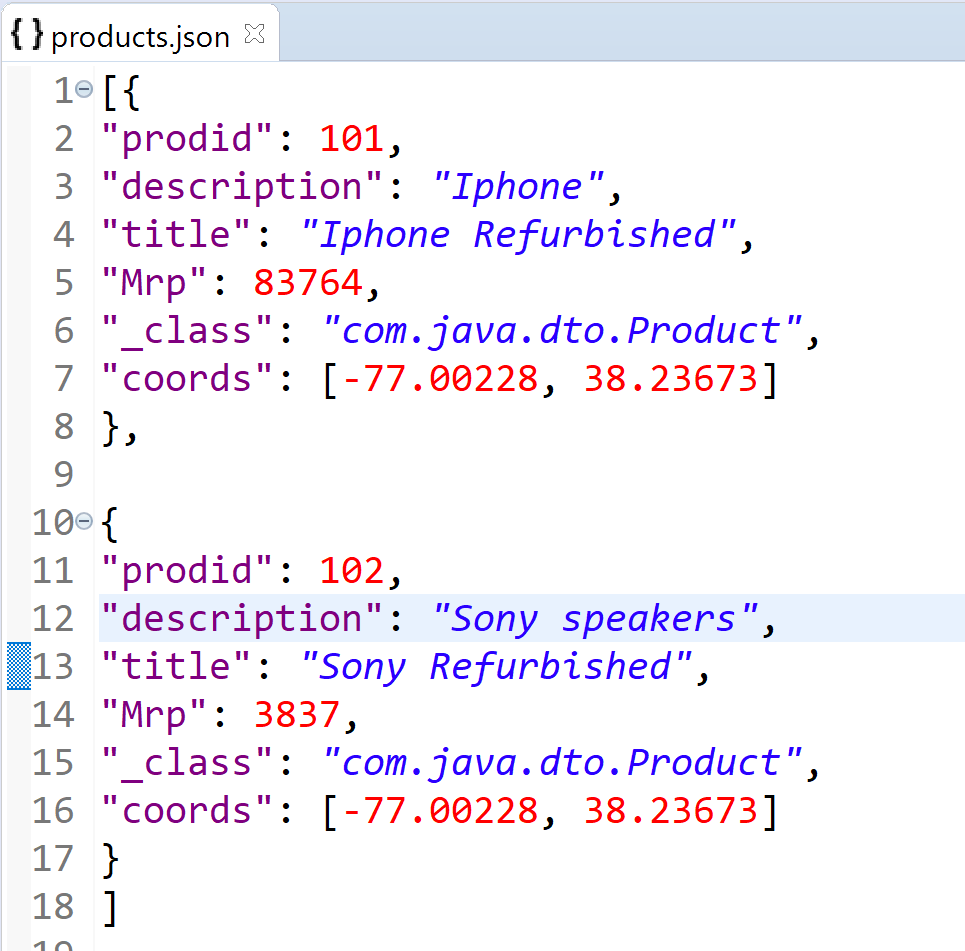
****

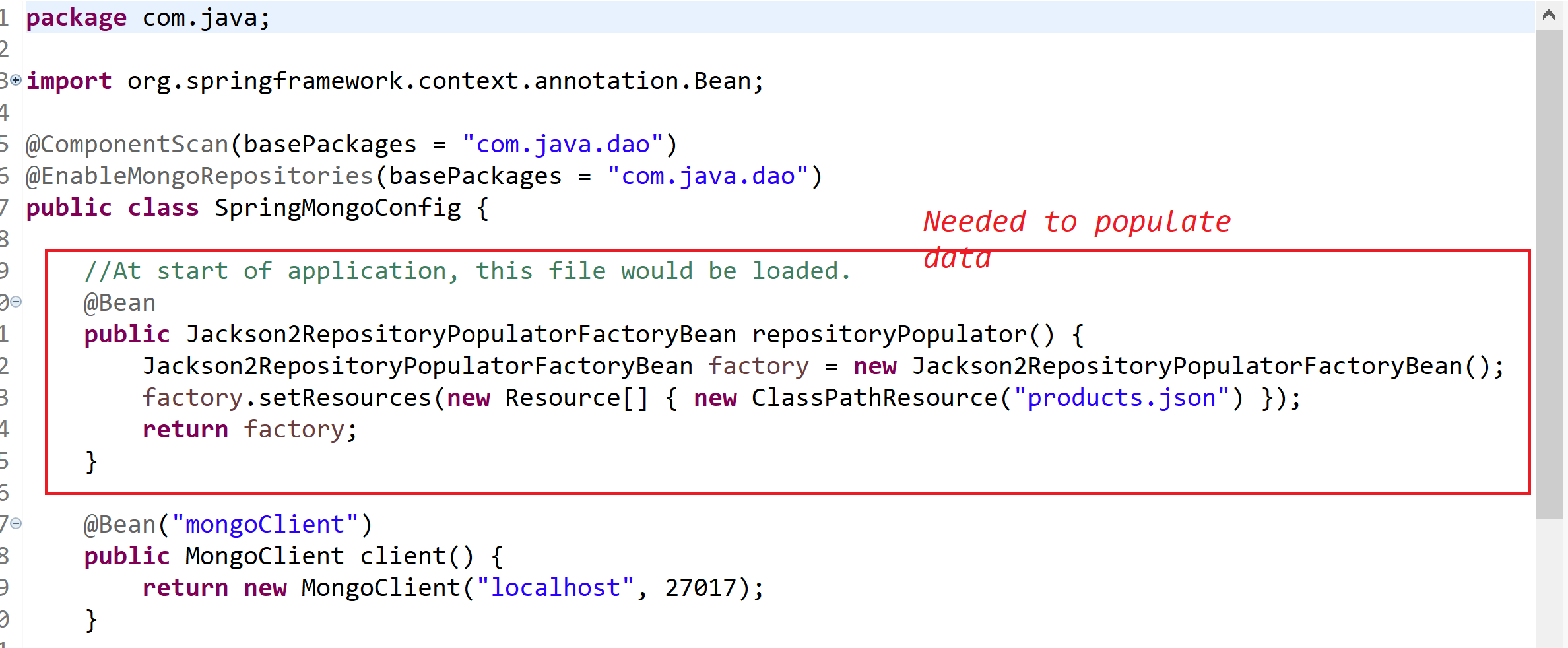
1. **MongoRespository:**

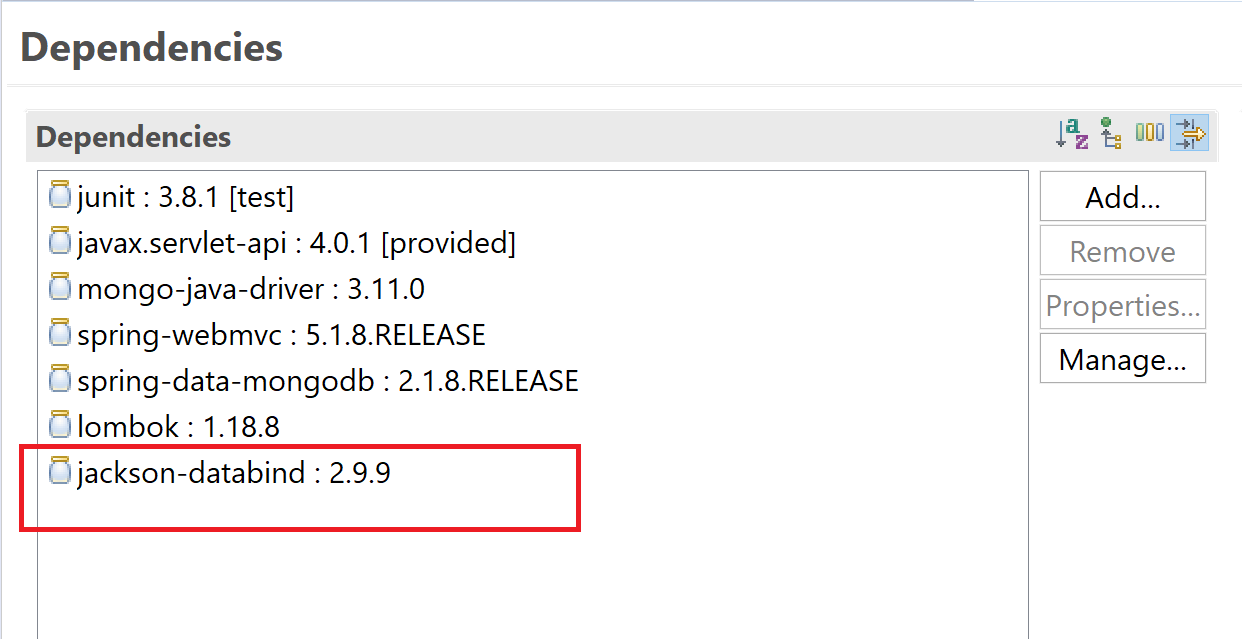




Populate data in db from json file:



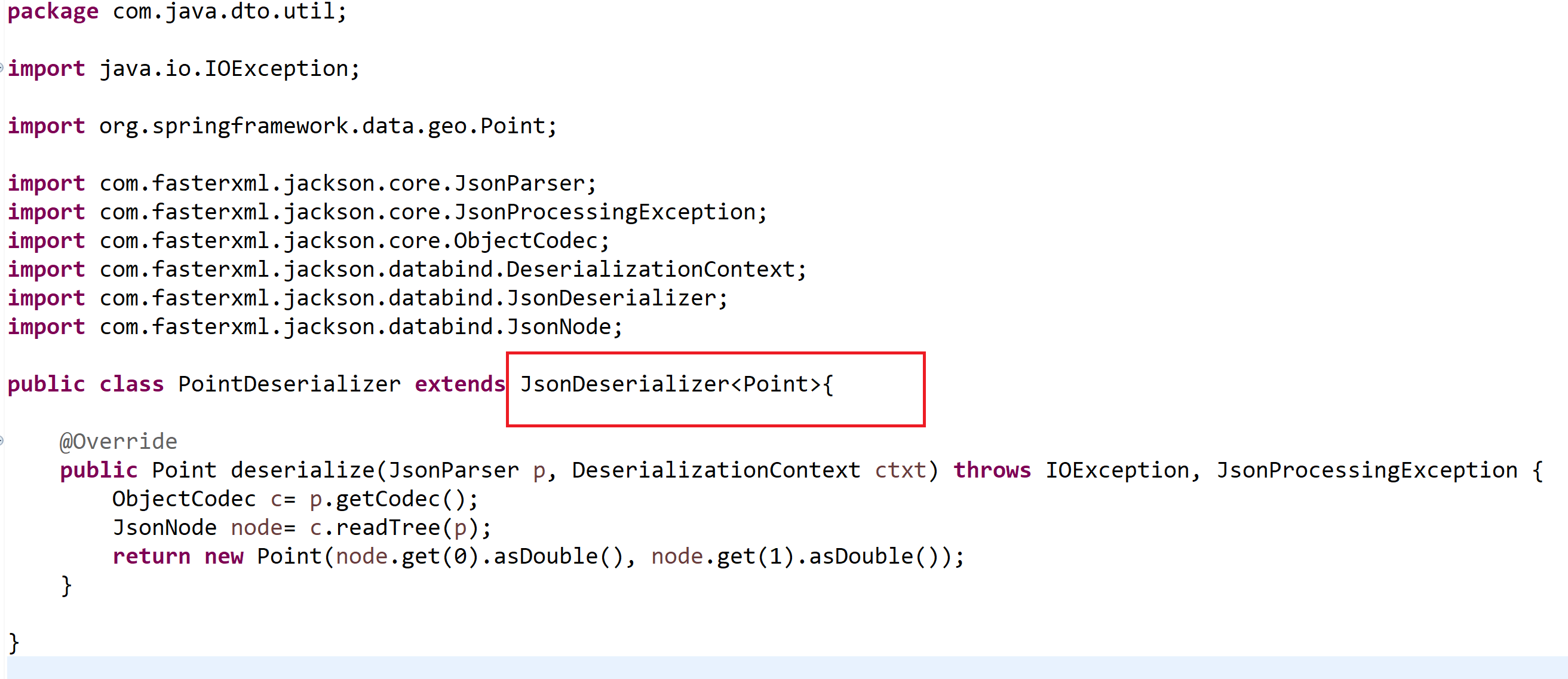


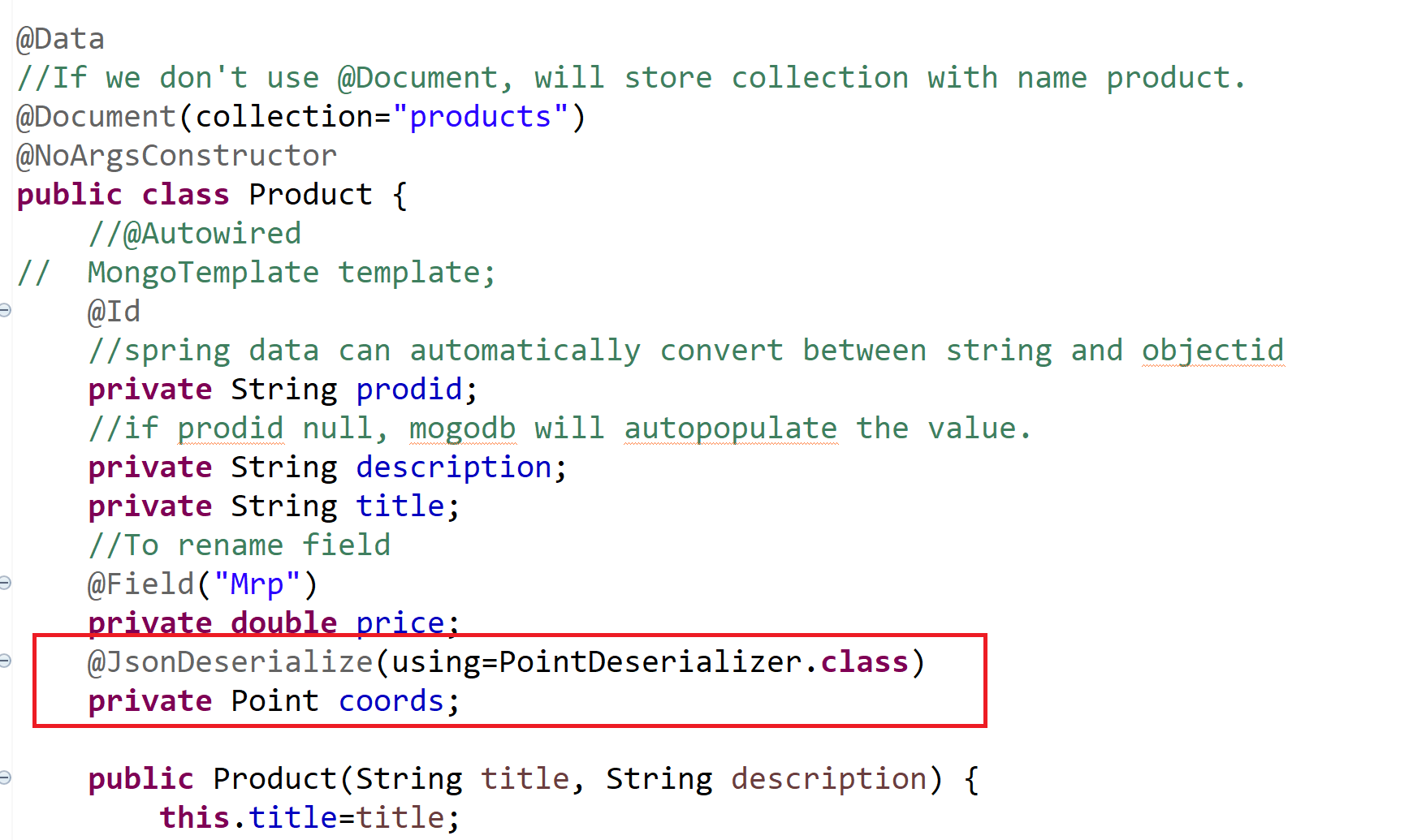


To deserialize json to java

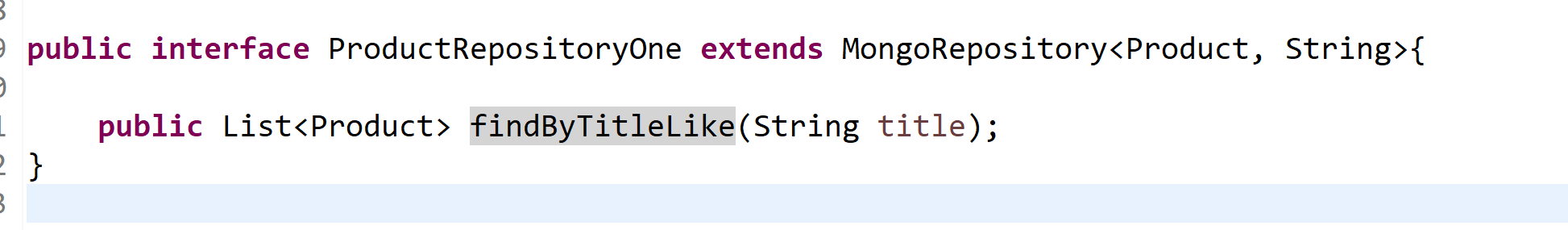
Product must have no-arg costurctor.

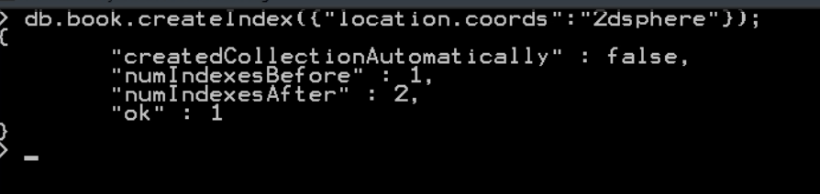
**Custom deserializer:**

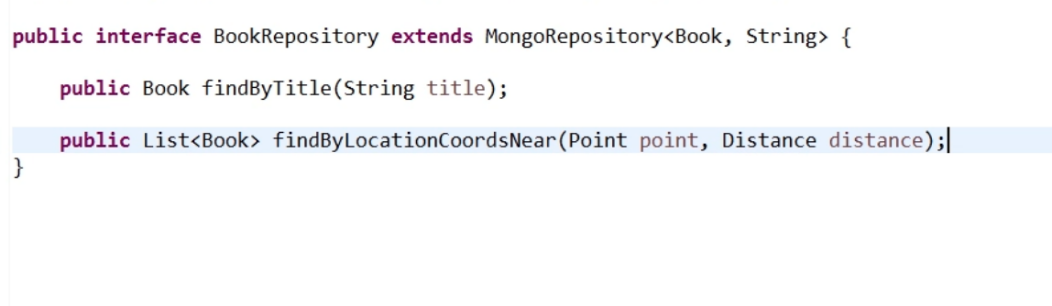


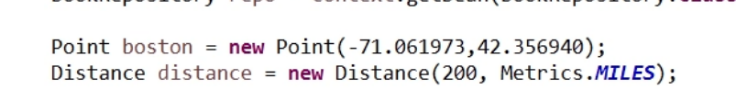


**Custom Methods:**

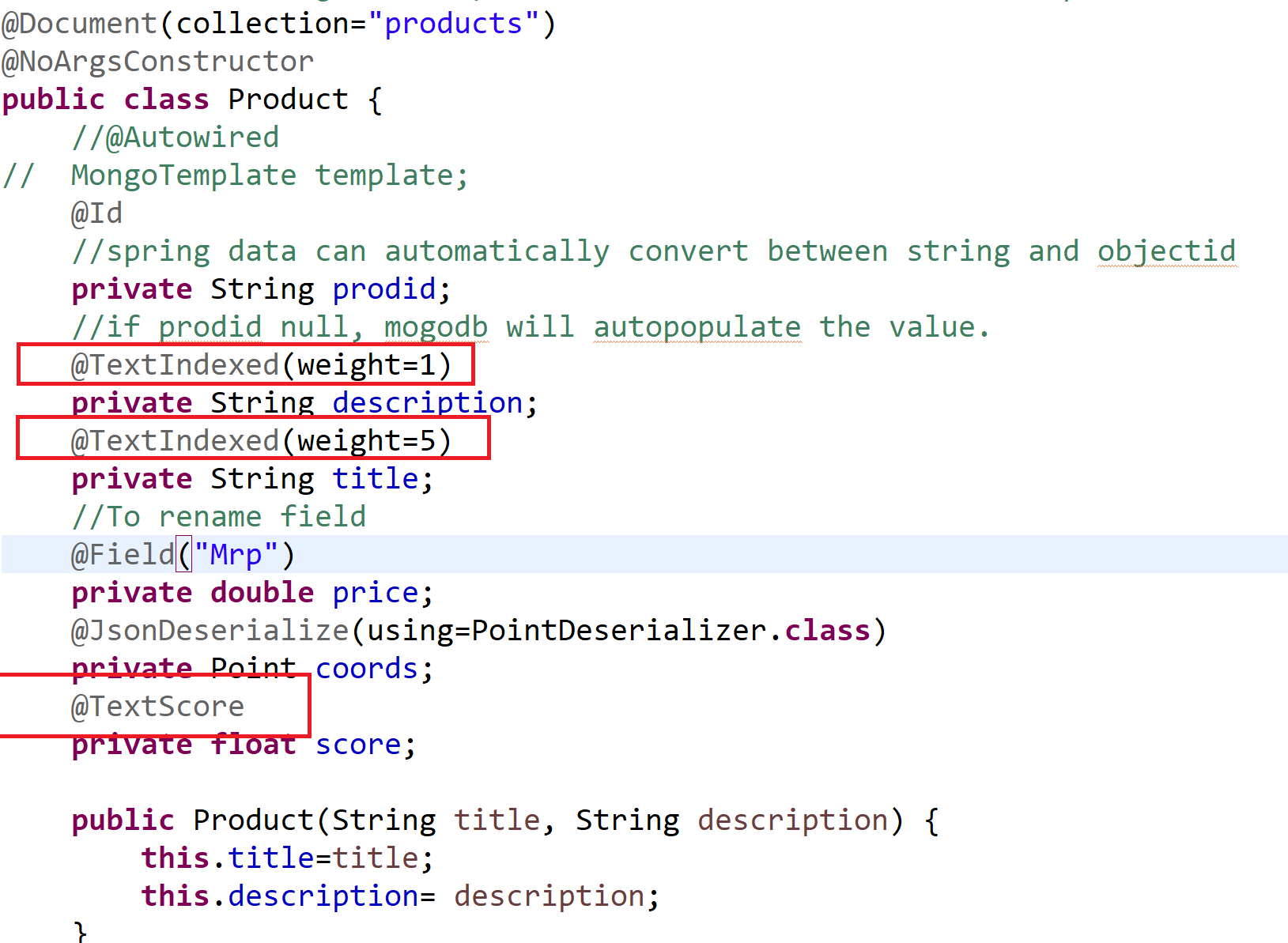


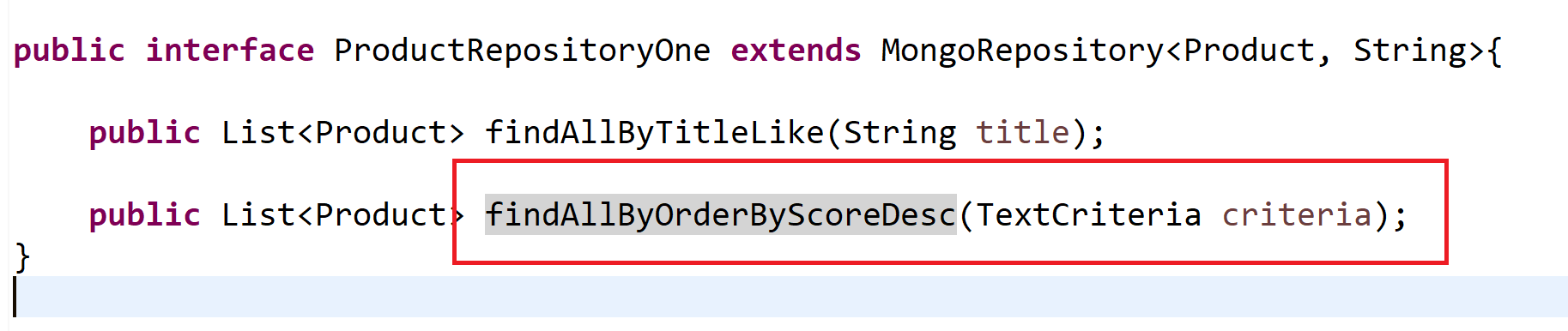


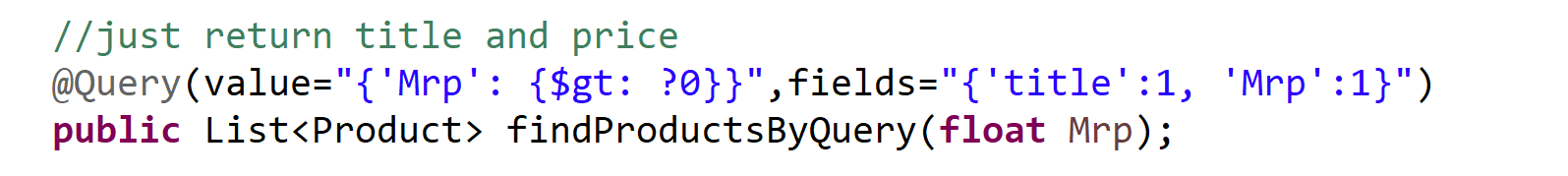




**Full text search:**





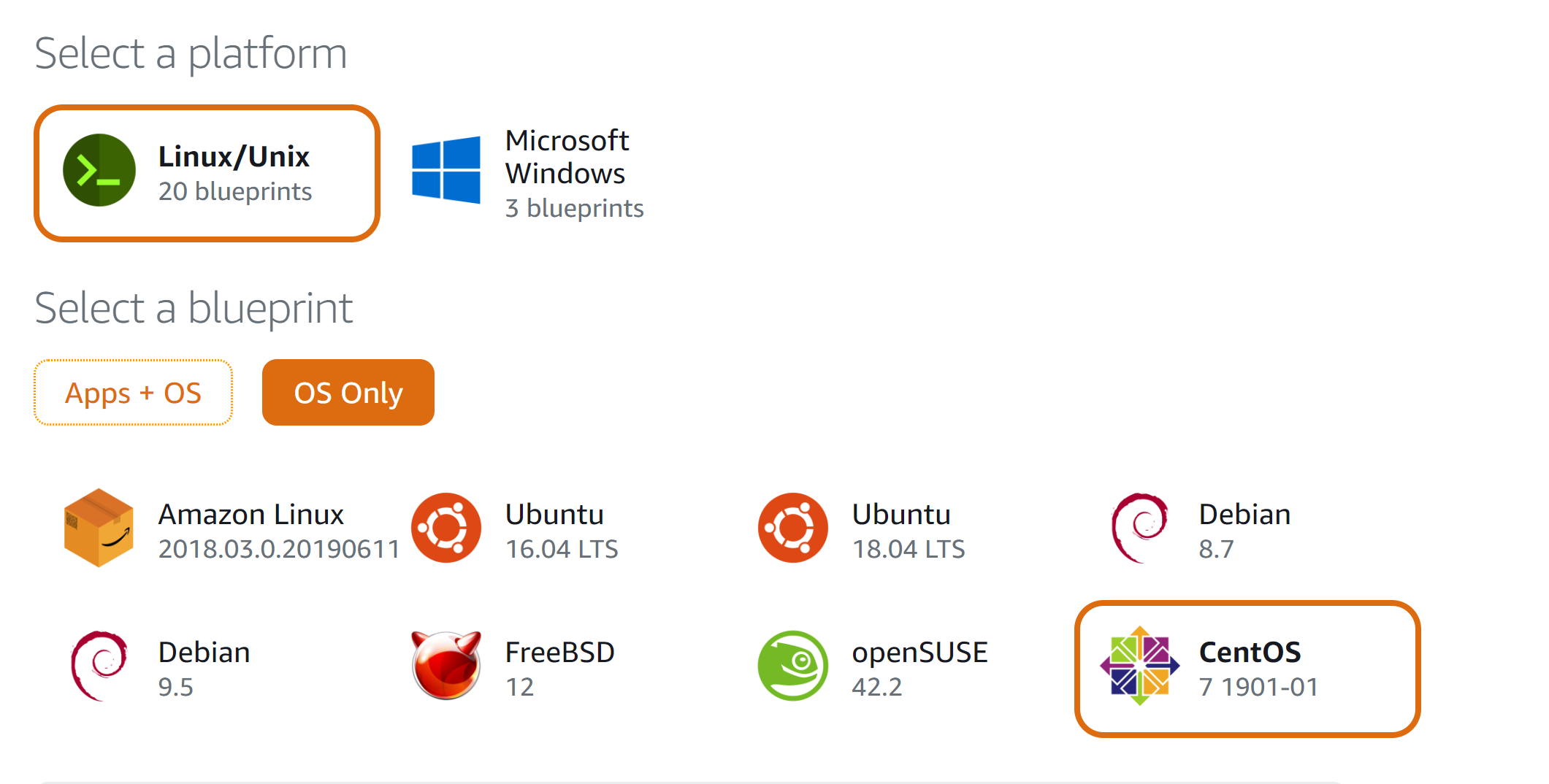


**Converter**



**Mongo cluster:**

Amazon lightsail:



Install mongodb:

[**Installing MongoDB**](https://linuxize.com/post/how-to-install-mongodb-on-centos-7/#installing-mongodb)

At the time of writing this article, the latest version of MongoDB available from the official MongoDB repositories is version 4.0. Before continuing with the next step visit the [Install on Red Hat](https://docs.mongodb.com/manual/tutorial/install-mongodb-on-red-hat/#configure-the-package-management-system-yum)section of MongoDB’s documentation and check if there is a new release available.

Follow the steps below to install the latest stable version of MongoDB on your CentOS server :

1. Enabling MongoDB repository

To add the MongoDB repository to your system, open your text editor and create a new YUM repository configuration file named mongodb-org.repo inside the /etc/yum.repos.d/ directory:

/etc/yum.repos.d/mongodb-org.repo

**[mongodb-org-4.0]**

name=MongoDB Repository

baseurl=https://repo.mongodb.org/yum/redhat/$releasever/mongodb-org/4.0/x86\_64/

gpgcheck=1

enabled=1

gpgkey=https://www.mongodb.org/static/pgp/server-4.0.asc

Copy

Sudo su

vi mongodb-org.repo

paste ur code

esc :wq! (write and exit)

*If you want to install an older version of MongoDB, replace each instance of 4.0 with your preferred version.*

1. Installing MongoDB

Now that the repository is enabled you can install the mongodb-org meta-package using the yum utility:

sudo yum install mongodb-org

Copy

During the installation yum will prompt you to import the MongoDB GPG key. Type y and hit Enter.

The following packages will be installed on your system as a part of the mongodb-org package:

* + mongodb-org-server - The mongod daemon, and corresponding init scripts and configurations.
  + mongodb-org-mongos - The mongos daemon.
  + mongodb-org-shell - The mongo shell, an interactive JavaScript interface to MongoDB, used to perform administrative tasks thought the command line.
  + mongodb-org-tools - Contains several MongoDB tools for to importing and exporting data, statistics, as well as other utilities.

1. Starting MongoDB

Once the installation is completed, start the MongoDB daemon and enable it to start on boot by typing:

sudo systemctl start mongod

sudo systemctl enable mongod

Copy

1. Verifying MongoDB Installation

To verify the installation we will connect to the MongoDB database server using the mongo tool and print the server version:

mongo

Copy

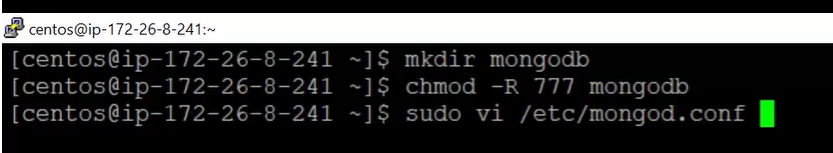
Once you are inside the MongoDB shell type the following command which will display the MongoDB version:

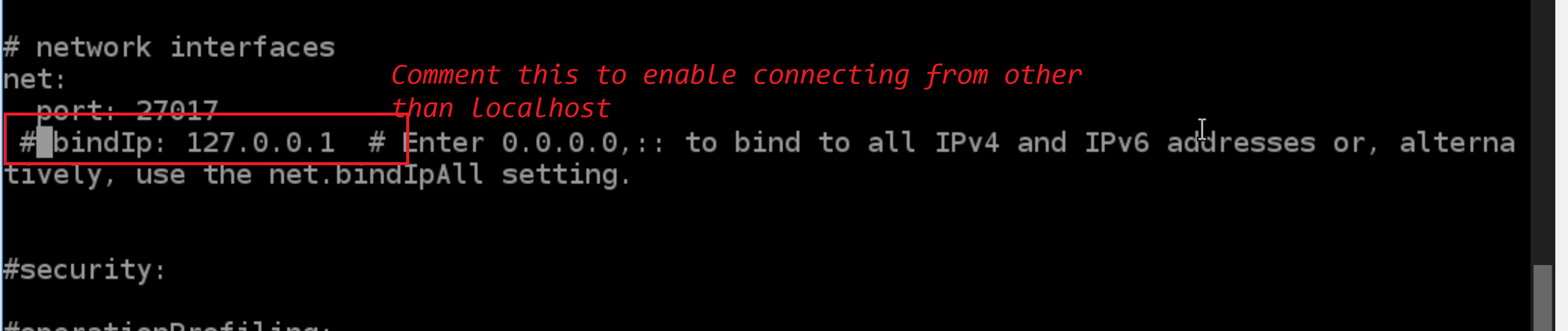
db.version()

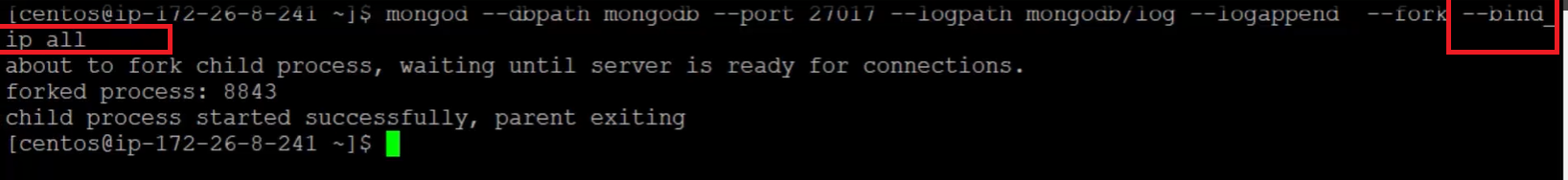
Copy

The output will look like the following:

4.0.1

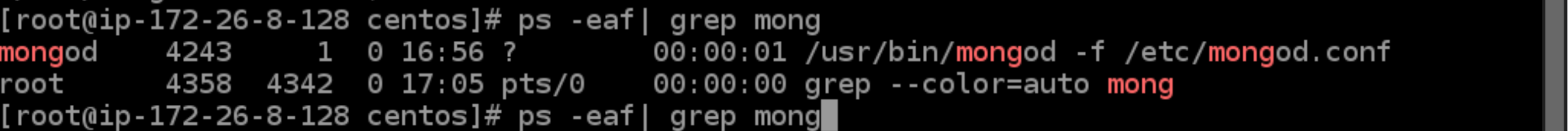




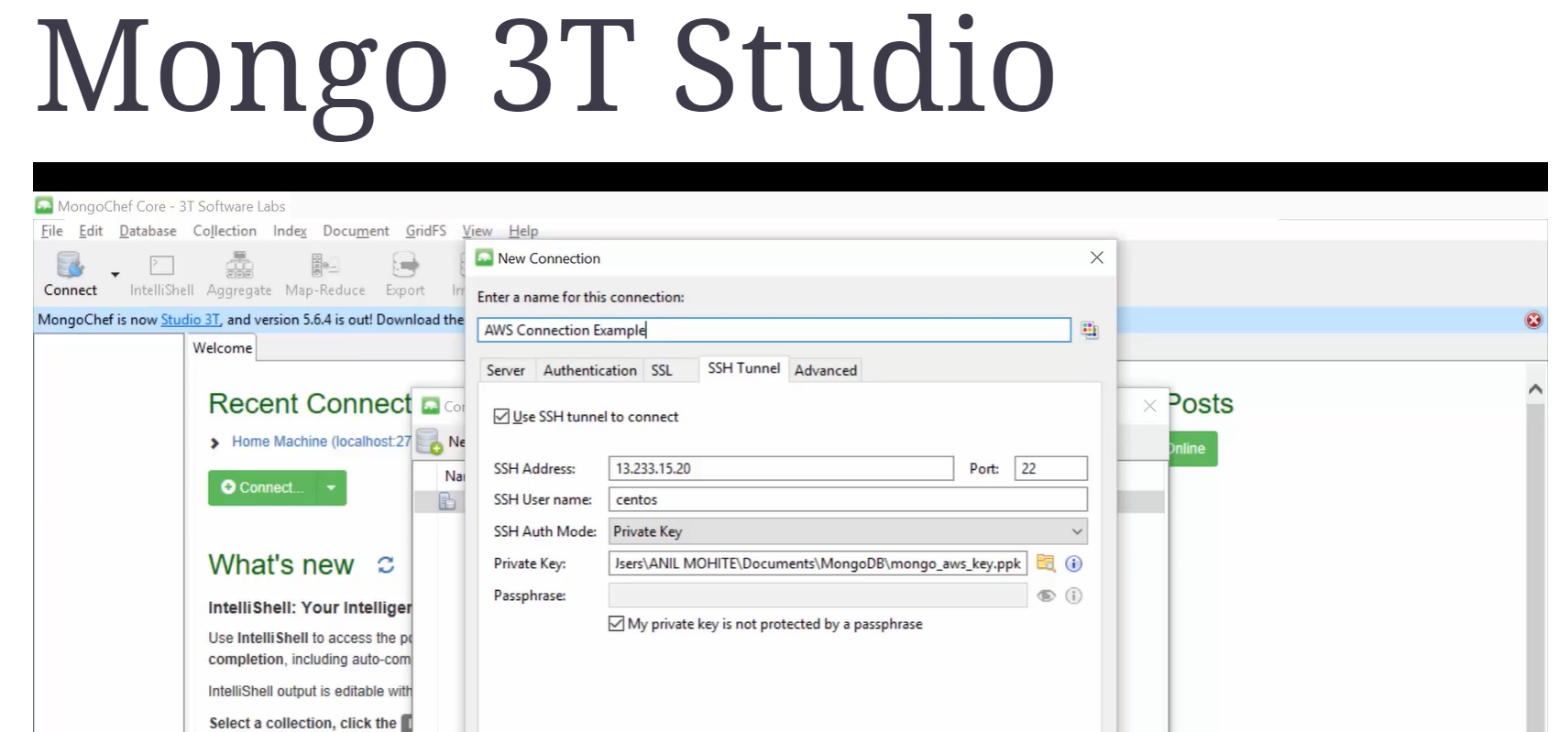


--bind\_ip\_all: to accept connections from everywhere

Mongo 3T Studio:



<https://hackernoon.com/how-to-install-and-secure-mongodb-in-amazon-ec2-in-minutes-90184283b0a1>

****

 @Bean  
    **public**MongoClient mongoClient() **throws**UnknownHostException {  
        MongoClientOptions.Builder builder =  **new**MongoClientOptions.Builder();  
        builder.connectionsPerHost(50);  
        builder.writeConcern(WriteConcern.***JOURNALED***);  
        builder.readPreference(ReadPreference.secondaryPreferred());  
        MongoClientOptions options = builder.build();  
        MongoClient mongoClient = **new**MongoClient(**new**ServerAddress(**env**.getProperty(**"mongo.server"**), Integer.*parseInt*(**env**.getProperty(**"mongo.port"**))), options);  
        **return**mongoClient;  
    }

### [2.1.1](https://www.blogger.com/null)[Single Field Index](https://www.blogger.com/null)

Single Field Index can be created by annotating a field of a domain object with @Indexed

**Avoid using unique index, and ensure uniqueness of the document at the application level.** It will improve performance, since MongoDB can avoid uniqueness check while inserting documents into the collection.

|  |
| --- |
| @Indexed  //by default the index direction is ASCENDING  **private**Long **employeeId**  @Indexed(direction = IndexDirection.***DESCENDING***) **private**DateTime **enrolledDateTime**; |

### 

### 2.1.2Compound Index

If your query is based on multiple keys, then construct a compound key, instead of making multiple single field indexes.

**Compound index needs to be constructed with fields in the following order:**

1.      Fields involved in Equality criteria

2.      Fields involved in Range criteria

@CompoundIndex(name = "slNo\_dt\_idx",def = "{'serialNumber' : 1, 'startDateTime' : 1, 'endDateTime' : 1}" )  
public class Employee {

….

}

# [1](https://www.blogger.com/null)[Best practices in configuration](https://www.blogger.com/null)

## 1.1      Connections Per Host

Number of physical connections that a mongo client can establish with the mongod process.

**By default:** 100

But can be increased/decreased based on our application requirement. It’s a good practice to identify the maximum number of connections to be available at the average load in your application - without keeping any threads waiting for available connections. The correct value will improve the performance and will help you to properly manage the resources (since each connection uses a certain RAM).

## [1.2      Connection Time Out](https://www.blogger.com/null)

Number of milliseconds a driver will wait before a connection attempt is failed.

**By default:** 10\*1000 milli seconds

In normal scenarios the driver will be able to make connection to the Mongod instance within a fraction of a second.

## [1.3      Threads Allowed To Block For Connection Multiplier](https://www.blogger.com/null)

Multiplier for connectionsPerHost that denotes the number of threads that are allowed to wait for connections to become available if the pool is currently exhausted. For example, if the connectionsPerHost is 100 (default) and this value is 5, then up to 500 threads can block before an exception is thrown.

**NOTE:**By setting the [**connectionsPerHost**](file:///C:\Users\310199867\Desktop\Spring%20Data%20MongoDB%20Best%20Practices.docx#_ConnectionsPerHost) to a correct value, we can reduce the number of threads waiting for an available connection.

## [1.4      Max Wait Time](https://www.blogger.com/null)

Number of milliseconds a thread can wait for a connection to get available in the connection pool if the pool is currently exhausted. And raise an exception if this does not happen in time.

**By default:** 1000 \* 60 \* 2 ms

## [1.5      Write Concern](https://www.blogger.com/null)

Based on this Mongo will decide whether to raise error or not. And controls the acknowledgment of write operations with various options.

**By default:** **WriteConcern.ACKNOWLEDGED**

In this option, the write operation will wait for acknowledgement from the primary server before returning. Will raise error/exception on network failures and server errors.

Probably it’s a good idea to set it to **WriteConcern.JOURNALED** – will wait for the server to group commit to the journal file on disk. – use this if we are worried about the durability. It will make sure none of your data (write operation data) is lost even if the mongod terminated due to a failure before writing to data files.

If we are not worried about the durability and only concerned about the failures- use **WriteConcern.SAFE** - will wait for acknowledgement from the primary server before returning. And it will raise error/exception on network as well as server failures.

Note: will have a little more performance impact compared to ACKNOWLEDGED since it sends a getLastError() command after your write operation – and until the lastError command is completed the connection is reserved.

**Preferred:**  **WriteConcern.JOURNALED**

***NOTE:****Starting with MongoDB 2.6 write operations will fail with an exception if this option is used when the server is running without journaling. If you are not worried about the outcome of the write operation, you can use writeConcern.UNACKNOWLEDGED (write operations return after it’s written to the socket – raise exception only on network failure) or WriteConcern.NONE (No exceptions are raised, even for network issues.)*

## [1.6      Read Preference](https://www.blogger.com/null)

Represents the preferred replica set members/nodes to which a query or command can be send.

It has different options:

**Default:** **ReadPreference.primary()** - all reads goes to the Primary member/node in the replica set.

   Note: use this if you want all reads to return consistent/ the most recently written data always.

**ReadPreference.primaryPreferred()** – All reads goes to primary member if possible but may query secondary members if primary is not available.

**ReadPreference.Seconday()** – all reads go to secondary members in the replica set and Primary member/node will be used for writes only. The reads become eventually consistent, because of the possible replication latency. More secondary nodes can be added to scale up the read performance, but there is a limit in the number of secondary nodes a replica set can have.

**ReadPreference.SecondaryPreferred()** - All reads go to Secondary nodes if any of them are available, if not then the reads will be routed to the Primary member of the replica set.

**ReadPreference.nearest()** – all reads go to the nearest replica set node/member to the client/application. Use only if eventually consistent reads are acceptable.

**Preferred:** If your requirement (for instance: ETL or analytical or Reporting) allows you for eventually consistent reads, then use **ReadPreference.SecondaryPreferred()**otherwise always use the default setting -**ReadPreference.primary()**

## [1.7      Set Mongo configurations using Spring Data](https://www.blogger.com/null)

|  |
| --- |
| @Configuration @PropertySource(value= **"classpath:/mongo.properties"**) @Profile({ **"default"**}) **public class**LocalMongoConfig {      @Autowired     Environment **env**;      @Bean     **public**MongoClient mongoClient() **throws**UnknownHostException {         MongoClientOptions.Builder builder =  **new**MongoClientOptions.Builder();         builder.connectionsPerHost(50);         builder.writeConcern(WriteConcern.***JOURNALED***);         builder.readPreference(ReadPreference.secondaryPreferred());         MongoClientOptions options = builder.build();         MongoClient mongoClient = **new**MongoClient(**new**ServerAddress(**env**.getProperty(**"mongo.server"**), Integer.*parseInt*(**env**.getProperty(**"mongo.port"**))), options);         **return**mongoClient;     }      @Bean     **public**MongoDbFactory mongoDbFactory() **throws**UnknownHostException {         MongoDbFactory mongoDbFactory = **new**SimpleMongoDbFactory(mongoClient(),  **env**.getProperty(**"mongo.databaseName"**),  **new**UserCredentials(**env**.getProperty(**"mongo.userName"**),  **env**.getProperty(**"mongo.password"**)));          **return**mongoDbFactory;      }      @Bean     **public**MongoTemplate mongoTemplate() **throws**UnknownHostException {         MongoTemplate mongoTemplate = **new**MongoTemplate(mongoDbFactory());         **return**mongoTemplate;     }   } |

**mongo.properties**

|  |
| --- |
| mongo.server=localhost mongo.port=27017 mongo.databaseName=Test mongo.userName= mongo.password= |

Note: All other configuration settings (MongoClientOptions) will be the default ones.

# [2](https://www.blogger.com/null)[Best Practices At Query Level](https://www.blogger.com/null)

## [2.1      Create Indexes](https://www.blogger.com/null)

Create indexes on frequently queried fields to avoid full collection scan and to improve performance.

### [2.1.1Single Field Index](https://www.blogger.com/null)

Single Field Index can be created by annotating a field of a domain object with @Indexed

**Avoid using unique index, and ensure uniqueness of the document at the application level.** It will improve performance, since MongoDB can avoid uniqueness check while inserting documents into the collection.

|  |
| --- |
| @Indexed  //by default the index direction is ASCENDING  **private**Long **employeeId**  @Indexed(direction = IndexDirection.***DESCENDING***) **private**DateTime **enrolledDateTime**; |

### 

### 2.1.2Compound Index

If your query is based on multiple keys, then construct a compound key, instead of making multiple single field indexes.

**Compound index needs to be constructed with fields in the following order:**

1.      Fields involved in Equality criteria

2.      Fields involved in Range criteria

|  |
| --- |
| Query query = **new**Query(**new**Criteria().andOperator(Criteria.*where*(**"serialNumber"**).is(serialNumber), **new**Criteria()         .orOperator(**new**Criteria().andOperator(Criteria.*where*(**"startDateTime"**).gte                 (startDateTime.withTimeAtStartOfDay().toDate()), Criteria.*where*(**"startDateTime"**).lt(endDateTime                 .toDate())), **new**Criteria().andOperator(Criteria.*where*(**"startDateTime"**).lt(startDateTime                 .withTimeAtStartOfDay().toDate()), Criteria.*where*(**"endDateTime"**).gt(startDateTime.toDate()))))); |

For this query to execute without a (full) collection scan, construct a compound index on the domain object Employee. Following compound index ensures an index scanning (o/p of explain() gives you cursorType as “BTreeCursor + “Index name”)

|  |
| --- |
| @CompoundIndex(name = "slNo\_dt\_idx",def = "{'serialNumber' : 1, 'startDateTime' : 1, 'endDateTime' : 1}" ) public class Employee {  ….  } |

Here serialNumber – involves in equality criteria; startDateTme, endDateTime – involve in Range criteria

The compound Index created in this order will give **n=nscanned=nscannedObjects** output when you execute explain()- means the best possible index to use.

        n – number of documents returned

        nscanned – number of indexes scanned

        nscannedObjects -  Number of documents scanned

## [2.2      Use Covered Indexes](https://www.blogger.com/null)

Try to use covered indexes if possible.

Covered indexes means the fields included in the result set and the fields used in the query are part of a single index. This will help mongo to return the result from index without scanning the document.

|  |
| --- |
| Query q1 = **new**Query(Criteria.*where*(**"code"**).in(**"abc"**,**"ijk"**, **"xyz"**)); q1.fields().include(**"name"**); |

@CompoundIndex(name = **"code\_name\_idx"**, def = **"{'code' : 1, 'name' : 1}"**)

## [2.3      Avoid long Field Names](https://www.blogger.com/null)

Avoid unnecessarily long field names. Field names are repeated across documents and consume space. Smaller field names allows for a larger number of documents to fit in RAM.

## [2.4      Include only Updated Fields in the Update Query](https://www.blogger.com/null)

Use **Update** object to issue updates to only modify fields that have changed, instead of retrieving the entire document in your application, updating fields and then saving the document back to the database.

For example, let’s say the salary of the employee with serialNumber 1009 has changed in that case instead of doing following:

|  |
| --- |
| Employee employee = mongoTemplate.findOne(**new**Query(Criteria.*where*(**"serialNumber"**).is(1009)), Employee.**class**); employee.setSalary(<updatedSalary>); mongoTemplate.save(employee, **"Employee"**); |

**Downside of the approach:** This issues – 2 Queries and sends entire object to just update the salary field.

In such cases we should use:

|  |
| --- |
| Update update = **new**Update() update.set(**"salary"**, <updatedSalary>); mongoTemplate.updateFirst(**new**Query(Criteria.*where*(**"serialNumber"**).is(1009)), update,  **"Employee"**); |

**Advantage:** In this case, it just issues one query and moreover just sends the updated field as part of the update.

## [2.5      Use Projections to Reduce the Amount of Data Returned](https://www.blogger.com/null)

Use Projections to avoid unnecessary data being returned to the application. Only include the necessary fields in the projection to include in the result set.

This can be achieved by:

|  |
| --- |
| Query query = **new**Query(Criteria.*where*(**"name"**).is(**"Adapter"**)); query.fields().include(**"code"**).include(**"tags"**)  List<Inventory> inventoryList = mongoTemplate.find(query,  Inventory.**class**); |

In this case, only two fields (code, tags) in the Inventory class is returned and other fields are ignored/excluded while returning.  This will improve the query latency.

## [2.6      Use bulk insert instead of individual inserts](https://www.blogger.com/null)

Its good practice to use bulk inserts using **insert(Collection<? extends Object> batchToSave, String collectionName)** method instead of using multiple individual inserts using **insert(Object objectToSave, String collectionName)** method.

It will reduce the number of trips to the database, so as the number of network trips.

## [2.7      Automatic Deletion of Documents From Collection](https://www.blogger.com/null)

If you have a requirement to remove documents from a collection that elapsed a certain time, then that can be achieved by setting TTL (Time To Live) option using an annotation **@Indexed(expireAfterSeconds= <no.of seconds>)**on a date field in the document

|  |
| --- |
| @Indexed(expireAfterSeconds = 604800) **private**DateTime **createdDateTime**; |

So in this case, the document will be automatically deleted from the collection by MongoDB after 7 days from the creation date time.

## [2.8    Use Capped Collections To Store Logs OR Small Caches](https://www.blogger.com/null)

It’s a good practice to create/use capped collections over normal collections to store documents for performing high throughput operations. Capped collections are fixed-size collections that support high-throughput operations that insert and retrieve documents based on insertion order.

            Create the Capped Collection by:

|  |
| --- |
| **private void**createCappedCollections(){     CollectionOptions options =  **new**CollectionOptions(100000, 50, **true**);     mongoTemplate.createCollection(AppLogger.**class**, options); } |

Note: You cannot create Capped Collections as normal Collections just specifying collection name in the @Document() annotation. In case of Capped collections, you need to specify the size, max number of documents etc. (CollectionOptions). And since capped collection ensures insert order, you don’t need to create indexes to get the documents in insert order from the collection.

Note: You can check whether the created collection is Capped collection or not by using

|  |
| --- |
| mongoTemplate.getCollection(<CollectionName>).isCapped() |

 More info: <http://docs.mongodb.org/manual/core/capped-collections/>

# [3](https://www.blogger.com/null)[AUDITING BEST PRACTICES](https://www.blogger.com/null)

## [3.1      Test every query in your application with explain()](https://www.blogger.com/null)

Spring Data MongoDB doesn’t provide a utility method for viewing query plan. But you could write a generic custom method, that uses explain() method from Mongo Java Driver using MongoTemplate to evaluate the query plan (for example, which index is used or whether full collection scan occurred etc.)

Include following method in the DAO class to monitor the query plan

|  |
| --- |
| **public void**performExplainQuery(Query query, String collectionName) {     DBCollection dbCollection = mongoTemplate.getCollection(collectionName);     DBCursor cursor = dbCollection.find(query.getQueryObject());     System.***out***.println(**"Query Plan: "**+ cursor.explain()); } |

Invoke this with two parameters - **query to evaluate** and the **collectionName on which the query needs to be executed**from your DAO methods.

## [3.2      Add Audit Entries to Model Objects](https://www.blogger.com/null)

Define an audit object with fields’ createdOn and updatedOn and version. Optionally can have createdBy and updatedBy if necessary. This will give information about when a document is last updated or created on. We found this very helpful while debugging and tracing the details.

|  |
| --- |
| **public class**Audit {     @CreatedDate     **private**DateTime **createdOn**;      @LastModifiedDate     **private**DateTime **updatedOn**;     @Version     **private**Long **version**;     **public**DateTime getCreatedOn() {         **return createdOn**;     }**public void**setCreatedOn(DateTime createdOn) {         **this**.**createdOn**= createdOn;     }     **public**DateTime getUpdatedOn() {         **return updatedOn**;     }     **public void**setUpdatedOn(DateTime updatedOn) {         **this**.**updatedOn**= updatedOn;     }     **public**Long getVersion() {         **return version**;     }     **public void**setVersion(Long version) {         **this**.**version**= version;     } } |
| @Document(collection=**"Employee"**) **public class**Employee **extends**Audit {  …  } |

On update of document (for instance: Employee), automatically **updatedOn** field will be updated and on create **createdOn** field will be populated.  **Version** field will be used for Optimistic Locking. It will be automatically incremented on update.

To get this working, you need to add the **@EnableMongoAuditing**annotation in the configuration class.

|  |
| --- |
| @Configuration @PropertySource(value= **"classpath:/mongo.properties"**) @Profile({ **"default"**,**"local-noproxy"**}) @EnableMongoAuditing **public class**LocalMongoConfig { |

Spring Data MongoDB will get the current user from session or spring security context based on the application settings to populate the createdBy and updatedBy fields.

## [3.3      Enable Optimistic Locking on Write Operations](https://www.blogger.com/null)

Implement Optimistic locking using @Version annotation. If two threads tried to update the same object at the same time, then one will be thrown will error saying the version is different than what the thread has. Because the first thread has incremented while updating the object. And so the other thread is actually working on stale data. So in that case, we can implement some kind of retry mechanism to make sure it can get the latest object and merge the changes before update into Mongo.

|  |
| --- |
| @Version     **private**Long **version**; |

## [3.4      Evaluate the Performance of each DAO Methods using Spring AOP](https://www.blogger.com/null)

It’s a good practice to see the performance of each DAO operation and see whether any special investigation is required to get the performance better. We can write a Spring Around Advise to do the same.

|  |
| --- |
| /\*\*  \* Aspect that implements automatic logging on performance of the data access queries with Spring Data MongoDB.  \*   \* **@author**Felix Jose  \*/ @Aspect @Component(**"PerformanceProfilerAspect"**) **public class**PerformanceProfilerAspect {       @Pointcut(**"execution(\* com.felix.dao.\*.\*(..))"**)     **public void**clientMethodPointcut() {     }     /\*\*     \* Log on the performance of the interactions/queries on MongoDB.     \*     \* **@param joinPoint**the join point     \* **@throws**Throwable the throwable     \*/    @Around(**"clientMethodPointcut()"**)    **public**Object retryOnConnectionException(ProceedingJoinPoint joinPoint) **throws**Throwable {         Object ret = **null**;           System.**out**.println(**"PerformanceProfilerAspect: Advised with logic to calculate the Time Taken for the    execution of the method ["**+joinPoint.getSignature()+**"]"**);                  StopWatch stopWatch = **new**StopWatch();         stopWatch.start();         String throwableName = **null**;             **try**{                        ret = joinPoint.proceed();              } **catch**(Throwable t) {                 throwableName = t.getClass().getName();                 **throw**t;             } **finally**{                 stopWatch.stop();                 **if**(throwableName != **null**) {                     System.**out**.println(**"Timed ["**+joinPoint.getSignature().toString()+**"]: "**+stopWatch                             .getTotalTimeMillis()+**" milliseconds , with exception ["**+throwableName+**"]"**);                 } **else**{                     System.**out**.println(**"Timed ["**+joinPoint.getSignature().toString()+**"]: "**+stopWatch                             .getTotalTimeMillis()+**" milliseconds"**);                 }             }          **return**ret;    } } |

Note: You can have logger implementation to log the performance statistics instead of System.out.println()

# [4](https://www.blogger.com/null)[Suggest Other Best Practices](https://www.blogger.com/null)

## [4.1   Upsert Without Update Object](https://www.blogger.com/null)

When you are not aware whether an object is already present in its mongo collection, then we should use upsert/save. But the problem with MongoTemplate.upsert() is that it expects an Update instance/object which should be populated with all the updated fields. But this can be tedious when the no. of fields in the domain Object is huge and the number of fields updated are unknown. In that case we cannot use MongoTemplate.upsert().

Now the other option is MongoTemplate.save() method, which accepts the domain object as its argument. And Mongo java driver will check whether the id field is present in the object or not. If the id is present then it’s considered for update otherwise the object is inserted. But the drawback of this approach is first we need to send a query to fetch the object from Mongo and then update its fields with the changesand then send it to the save method. So there are two db calls.

The best approach is using mongoTemplate.execute() method as follows:

|  |
| --- |
| **public boolean**persistEmployee(Employee employee) **throws**Exception {      BasicDBObject dbObject = **new**BasicDBObject();     mongoTemplate.getConverter().write(employee, dbObject);     mongoTemplate.execute(Employee.**class**, **new**CollectionCallback<Object>() {         **public**Object doInCollection(DBCollection collection) **throws**MongoException, DataAccessException {             collection.update(**new**Query(Criteria.*where*(**"name"**).is(employee.getName())).getQueryObject(),                     dbObject,                     **true**,  *// means upsert - true***false***// multi update – false*);             **return null**;         }     });     **return true**; } |

 This gives the flexibility to avoid making separate call to mongo and the updating the object received from mongo with the actual changes and then send that updated object to update into mongo..

## [4.2    Don’t Use ID field in Domain Objects](https://www.blogger.com/null)

If your domain object has the id field or any field annotated with @Id, and this object is involved in upsert/ MongoTemplate.execute with collection.update having upsert option true, then MongoDB java driver will insert if the document is not present or update if the document is present while the given query is executed.

But the inserted document will have \_id populated as null, if you application didn’t assign any value to it.

|  |
| --- |
| @Document(collection = **"Employee"**) **public class**Employee {     *@Id     private ObjectId id;*@Indexed     **private**String **serialNumber**;     @NotNull       ……  } |

|  |
| --- |
| BasicDBObject dbObject = **new**BasicDBObject(); **sdcMongoTemplate**.getConverter().write(employee, dbObject);  **sdcMongoTemplate**.execute(Employee.**class**, **new**CollectionCallback<Object>() {     @Override     **public**Object doInCollection(DBCollection collection) **throws**MongoException, DataAccessException {         collection.update((**new**Query(Criteria.*where*(**"serialNumber"**).is(serialNumber))).getQueryObject(),                 dbObject, **true**, **false**);         **return null**;     } }); |

Creates a document for the first time with \_id = null

|  |
| --- |
| { "\_id" : null, "\_class" : "com.felix.Employee", "serialNumber" : "15050803",….. |

But if you comment out the id field from the domain class, MongoDB java driver will automatically populate \_id field in the document while inserting.

|  |
| --- |
| @Document(collection = **"Employee"**)  **public class** Employee {  */\*@Id*  *private ObjectId id;\*/*  @Indexed     **private**String **serialNumber**;     @NotNull       ……  } |

|  |
| --- |
| BasicDBObject dbObject = **new**BasicDBObject(); **sdcMongoTemplate**.getConverter().write(employee, dbObject);  **sdcMongoTemplate**.execute(Employee.**class**, **new**CollectionCallback<Object>() {     @Override     **public**Object doInCollection(DBCollection collection) **throws**MongoException, DataAccessException {         collection.update((**new**Query(Criteria.*where*(**"serialNumber"**).is(serialNumber))).getQueryObject(),                 dbObject, **true**, **false**);         **return null**;     } }); |

|  |
| --- |
| { "\_id" : ObjectId("562138337ae8629d987f01e6"), "\_class" : "com.felix.Employee", "serialNumber" : "15050803",….. |

## [4.3    Remove Unnecessary \_Class Field](https://www.blogger.com/null)

 Spring MongoDB by default includes \_class field pointing to the entity's fully-qualified class name in the document as some kind of hint about what type to instantiate actually. Since mongo collection can contain documents that represent instances of a variety of types.

E.g.:  store a hierarchy of classes (inheritance)

In case you want to avoid writing the entire Java class name as type information but rather like to use some key you can use the @TypeAlias annotation at the entity class being persisted.

|  |
| --- |
| @Document(collection = **"Inventory"**) @TypeAlias(**"Inventory"**) **public class**Inventory { |

But when you retrieve documents using MongoTemplate.find..() by providing/passing the actual entity object type to which the document to be converted (eg: MongoTemplate.find(new Query(), **Inventory.class**)), then you may not need this \_class field in the corresponding document at all. So in that case, we can avoid spring mongo to include \_class field in each document by defining:

|  |
| --- |
| @Bean **public**MongoTemplate mongoTemplate() **throws**UnknownHostException {     MappingMongoConverter mappingMongoConverter =  **new**MappingMongoConverter(**new**DefaultDbRefResolver             (mongoDbFactory()), **new**MongoMappingContext());     mappingMongoConverter.setTypeMapper(**new**DefaultMongoTypeMapper(**null**));     **return new**MongoTemplate(mongoDbFactory(), mappingMongoConverter ); } |

This will help us to reduce the document size, since \_class will be there in all the documents and consume space.

## Manually build text indexes

@Document

class CookingRecipe {

String title;

String content;

}

Here we’ve got a very simple entity CookingRecipe and want to have a text index based on its title and content fields, putting some weight of 2 to search hits in title. Attaching weights to fields allows you to influence relevancy of a document when being looked up. It defines the significance of a field relative to others, boosting the documents score. In this case it doubles the documents relevance when hitting a match in title. The raw MongoDB index definition would look something like this:

{

title : "text",

content : "text"

},

{

weights: { title : 2 }

}

As of version 1.5 M1 of Spring Data MongoDB we can create a text index, capturing the fields we want to have full text search enabled on, manually.

TextIndexDefinition textIndex = new TextIndexDefinitionBuilder()

.onField("title", 2F)

.onField("content")

.build();

MongoTemplate template = … // obtain MongoTemplate

template.indexOps(CookingRecipe.class).ensureIndex(textIndex);

Alternatively, let the index be created automatically using mapping annotations. All we need to do is adding a few hints on the domain class and we are good to go.

@Document

class CookingRecipe {

@TextIndexed(weight=2) String title;

@TextIndexed String content;

}

Please note that it is only possible to have one full text index per collection. Now that we’ve created the index we’ll query the top 5 recipes matching “coffee” or “cake”.

TextCriteria criteria = TextCriteria.forDefaultLanguage()

.matchingAny("coffee", "cake");

Query query = TextQuery.queryText(criteria)

.sortByScore()

.with(new PageRequest(0, 5));

List<CookingRecipe> recipes = template.find(query, CookingRecipe);

Note that we provide dedicated types TextCriteria and TextQuery to express searches in full detail.

## Scoring

As mentioned earlier, documents get scored while searching. The score value is not returned by default, but since this information is often helpful we can include it in the output by adding { score : { $meta : "textScore" } } to the projection, which is implicitly done by calling query.sortByScore(). To access the score in the resulting documents we add a @TextScoreannotated property to CookingRecipe.

@Document

class CookingRecipe {

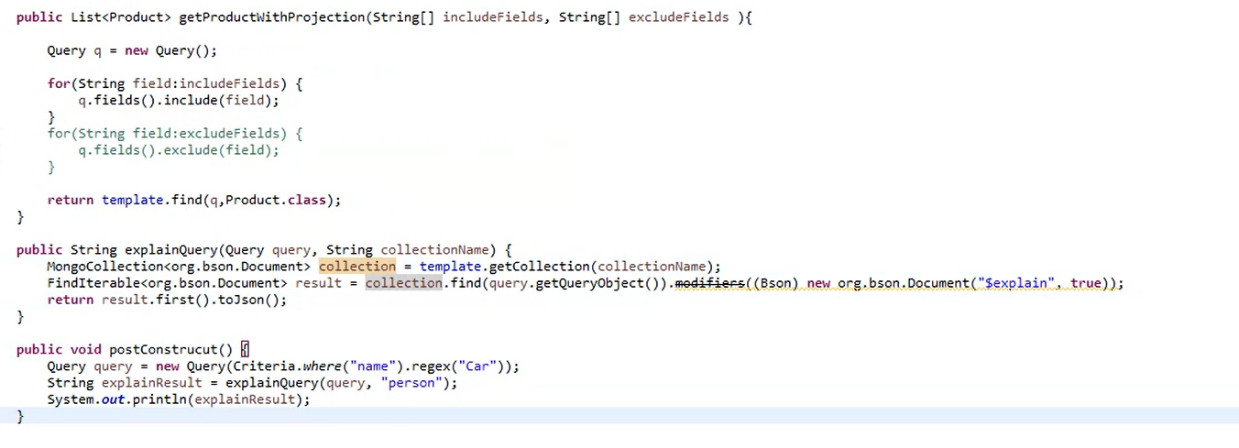
@TextIndexed(weight=2) String title;

@TextIndexed String content;

@TextScore Float score;

}

The @TextScore annotation implicitly turns the score property into a read-only property due to the @ReadOnlyProperty annotation the annotation carries in turn. The latter can be used in other contexts as well where you’d like make sure fields from a document are only read but never written.



Mongodb replication

Spring: Spring Data Repositories: Interfaces: On compilation: add code to ur project

So it reduces ur work

Custom Methods :

@EnableMongoRepositories (basePackages=”com.java.dao”, repositoryImplementationPostfix=”Implementation”)

//default is Impl

Class SpringConfig{  
}

Interface I1 extends MongoRespository, I2{

}

Interface I2{

Void update(String title);

}

Class I1Implementation implements I2{

Void update(String title){}

}