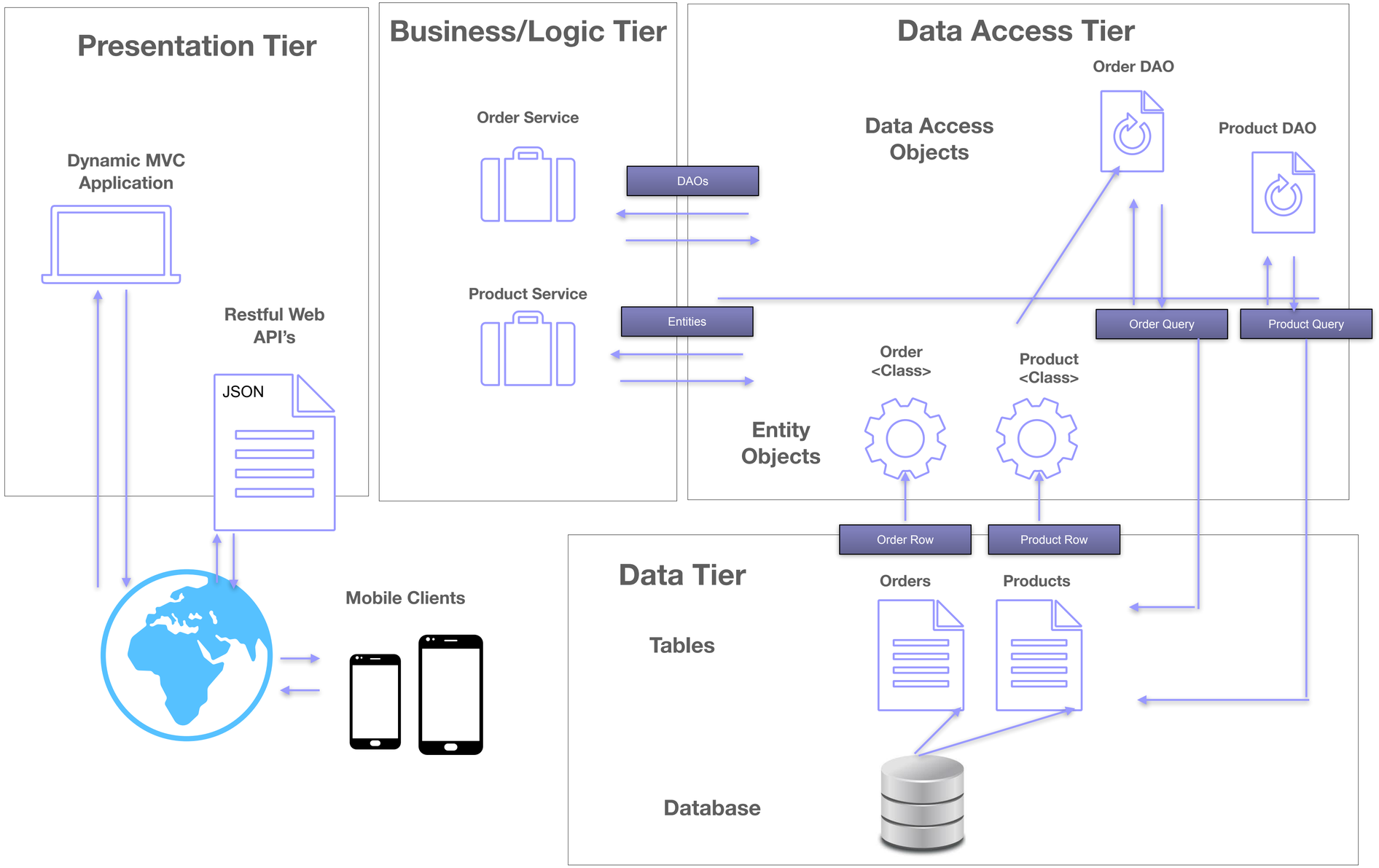
Spring Data and MongoDB

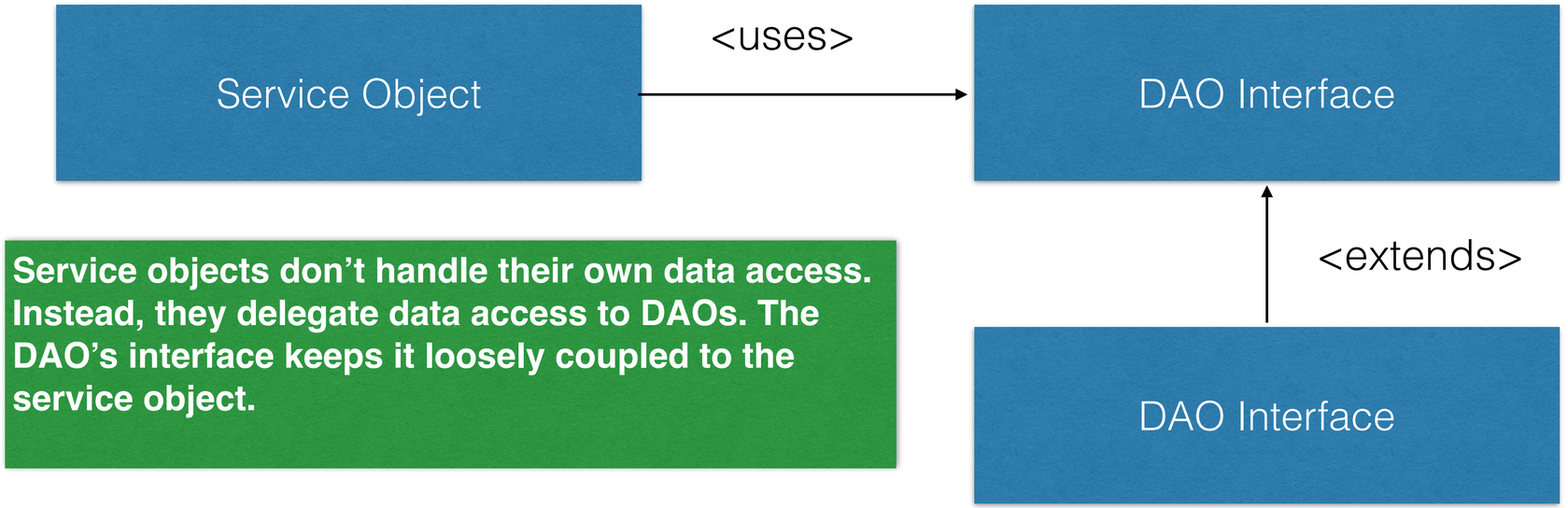
MongoDB Applications on Spring Boot

Multi-tier Backend



Spring Data Access Philosophy

- DAOs exist to provide a means to read and write data to the database.
- They should expose this functionality through an interface by which the rest of the application will access them



Working with Data

- Spring comes with a family of data access frameworks that integrate with a variety of data access technologies
  - JDBC
  - ORM (JPA, Hibernate, …)
  - MongoDb
  - …
- Aim is to isolate what happens at the data layer from the rest of the application

Spring Data

- DAO layer in Spring Framework is managed by Spring Data Repositories.
- The central interface in the Spring Data repository abstraction is Repository. It takes the domain class to manage as well as the ID type of the domain class as type arguments.
- The CrudRepository interface provides sophisticated CRUD functionality for the entity class that is being managed.

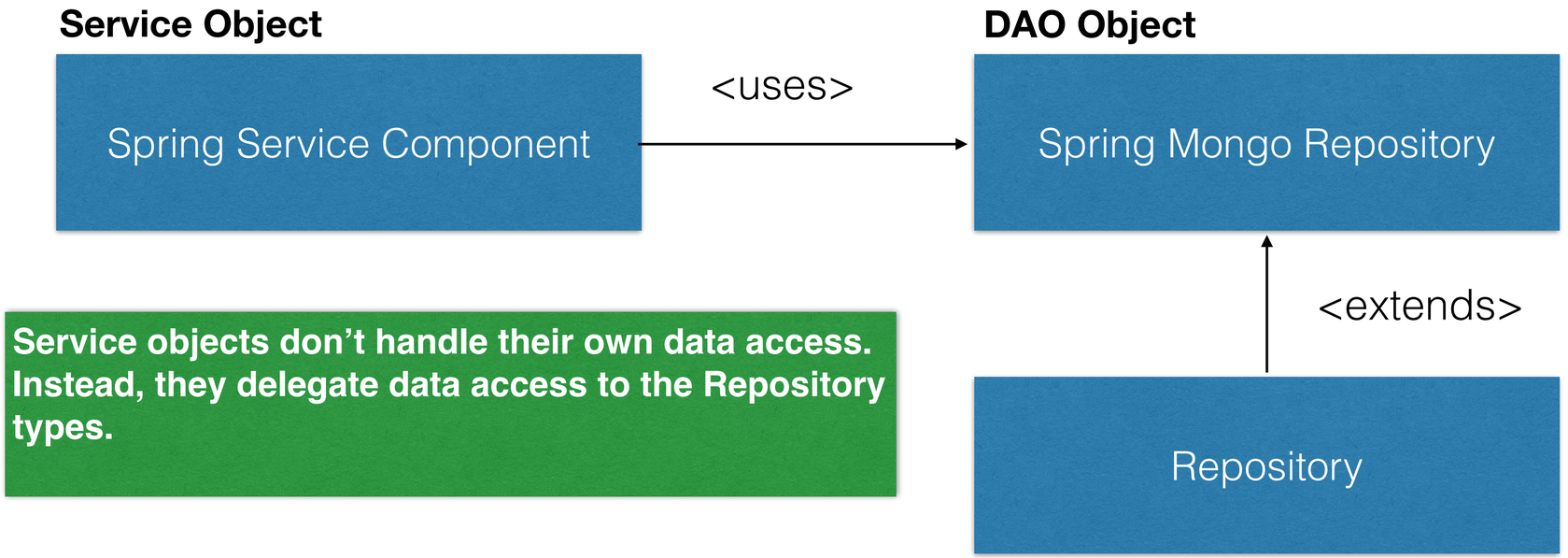
```
public interface CrudRepository<T, ID> extends Repository<T, ID> {

  <S extends T> S save(S entity);       1

  Optional<T> findById(ID primaryKey); 2

  Iterable<T> findAll();               3

  long count();                        4

  void delete(T entity);               5

  boolean existsById(ID primaryKey);    6

  // … more functionality omitted.
}
```

1. Saves the given entity.
2. Returns the entity identified by the given ID.
3. Returns all entities.
4. Returns the number of entities.
5. Deletes the given entity.
6. Indicates whether an entity with the given ID exists.

Spring Data Repositories

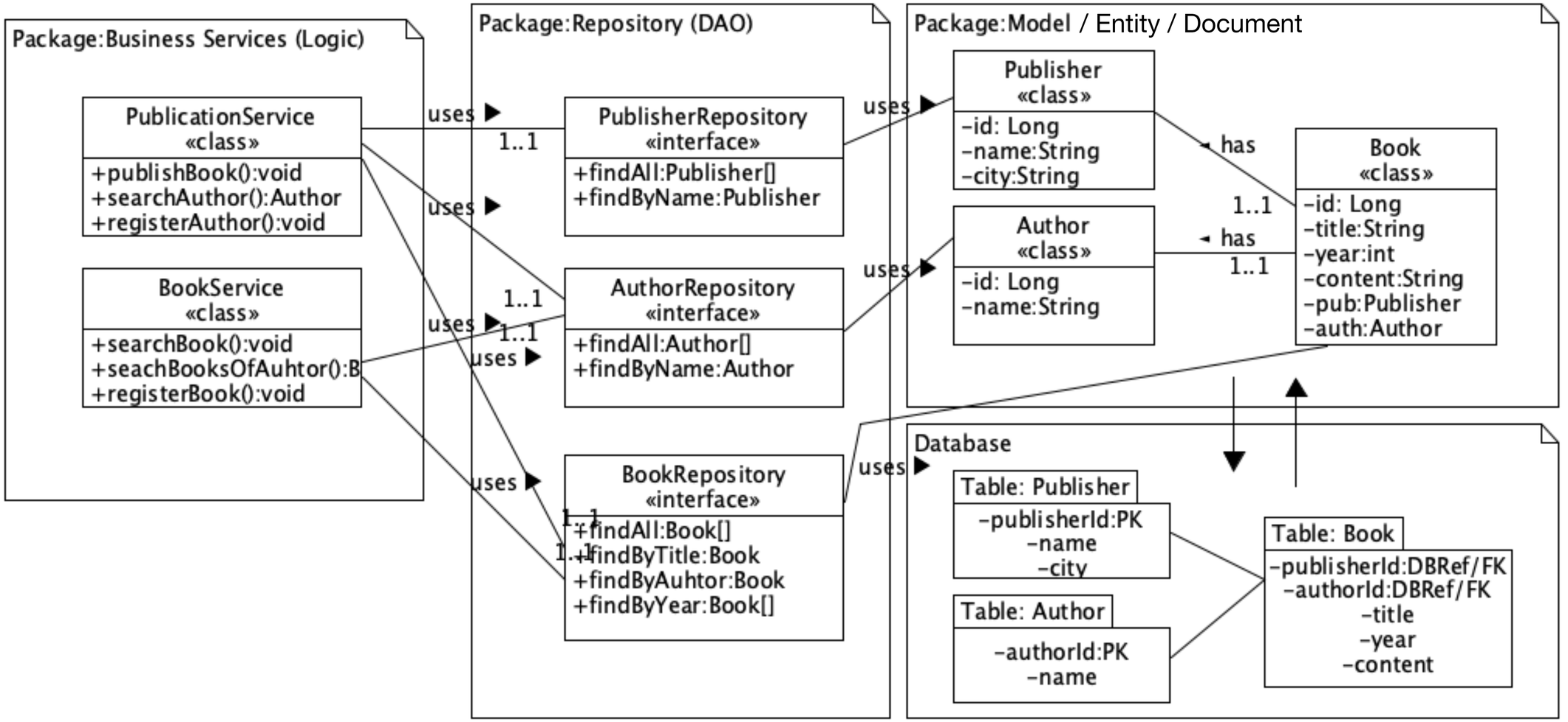
- Spring Repository implementations are provided for DAO functionality

- @Document (MongoDB), @Entity (JPA) are model type classes.



An Example

- A publisher requests a web application to keep track of their books and authors



Spring Data Implementations

- Persistence technology-specific abstractions, such as JpaRepository or MongoRepository are implemented.
- Those interfaces extend CrudRepository and expose the capabilities of the underlying persistence technology in addition to the rather generic persistence technology-agnostic interfaces such as CrudRepository.
- On top of the CrudRepository, there is a PagingAndSortingRepository abstraction that adds additional methods to ease paginated access to entities:

```java
public interface PagingAndSortingRepository<T, ID> extends CrudRepository<T, ID> {

  Iterable<T> findAll(Sort sort);

  Page<T> findAll(Pageable pageable);
}
```

To access the second page of `User` by a page size of 20, you could do something like the following:

```java
PagingAndSortingRepository<User, Long> repository = // … get access to a bean
Page<User> users = repository.findAll(PageRequest.of(1, 20));
```

How to Use Spring Boot with MongoDB

Spring Boot creates quick production-ready applications. MongoDB and Spring Boot interact using the MongoTemplate class and MongoRepository interface.

- **MongoTemplate** — MongoTemplate implements a set of ready-to-use APIs. A good choice for operations like update, aggregations, and others, MongoTemplate offers finer control over custom queries.

- **MongoRepository** — MongoRepository is used for basic queries that involve all or many fields of the document. Examples include data creation, viewing documents, and more.

- Spring Boot MongoDB configuration using both approaches needs only a few lines of code.

Configuring Spring Data MongoDB

- Add Maven dependency for Spring Boot Starter MongoDB:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

- Add @EnableMongoRepositories annotation after @SpringBootApplication
- This annotation helps discovery of MongoDB Repository classes

```
@SpringBootApplication
@EnableMongoRepositories
public class MdbSpringBootApplication implements CommandLineRunner{

    @Autowired
    ItemRepository groceryItemRepo;

    public static void main(String[] args) {
        SpringApplication.run(MdbSpringBootApplication.class, args);
    }
```

Configuring Spring Data MongoDB

- Navigate to resources folder and add MongoDB connection string in application.properties file:

```
spring.data.mongodb.uri=mongodb://[USERNAME]:[PASSWORD]@localhost:27017
spring.data.mongodb.database=books
```

- There is no need to write connection-related code in any other file. Spring Boot takes care of the database connection for us.

- We are also specifying the database name here — if it doesn't exist, MongoDB will create one.

MongoDB Model Implementation

- Our model here is the POJO or the GroceryItem class.
- We use the annotation **@Document** to set the collection name that will be used by the model. If the collection doesn’t exist, MongoDB will create it.
- The field level **@Id** annotation can decorate any type, including long and string
- If the value of the @Id field is not null, it's stored in the database as-is; otherwise, the converter will assume we want to store an ObjectId in the database (either ObjectId, String or BigInteger work).

```
@Document("groceryitems")
public class GroceryItem {

        @Id
        private String id;

        private String name;
        private int quantity;
        private String category;

        public GroceryItem(String id, String name, int quantity, String category) {
            super();
            this.id = id;
            this.name = name;
            this.quantity = quantity;
            this.category = category;
        }
}
```

API Implementation

- The API implementation happens in the repository. It acts as a link between the model and the database, and has all the methods for CRUD operations.
- We first create an ItemRepository public interface which extends the MongoRepository interface, then we can autowire the repository to any Spring controlled component.

```
public interface ItemRepository extends MongoRepository<GroceryItem, String> {

    @Query("{name:'?0'}")
    GroceryItem findItemByName(String name);

    @Query(value="{category:'?0'}", fields="{'name' : 1, 'quantity' : 1}")
    List<GroceryItem> findAll(String category);

    public long count();

}
```

```
@SpringBootApplication
@EnableMongoRepositories
public class MdbSpringBootApplication implements CommandLineRunner{

    @Autowired
    ItemRepository groceryItemRepo;

    public static void main(String[] args) {
        SpringApplication.run(MdbSpringBootApplication.class, args);
    }
```

Read Operations - Generated Methods

findAll()
findByX(X)
findByXandY(X,Y)
findByXGreaterThan(value)
findByXLessThen(value)
findByXGreaterThanOrEqualTo(value)
findByXLessThenOrEqualTo(value)

```
List<User> findByName(String name);
```

```
List<User> users = userRepository.findByName("Eric");
```

StartingWith and EndingWith
Between
Like and OrderBy

```
List<User> findByNameStartingWith(String regexp);
List<User> findByNameEndingWith(String regexp);
```

```
List<User> users = userRepository.findByNameStartingWith("A");
List<User> users = userRepository.findByNameEndingWith("c");
List<User> users = userRepository.findByAgeBetween(20, 50);
List<User> users = userRepository.findByNameLikeOrderByAgeAsc("A");
```

Paging Query Results

```
Pageable pageableRequest = PageRequest.of(0, 1);
Page<User> page = userRepository.findAll(pageableRequest);
List<User> users = pages.getContent();
```

Read Operations - JSON Query Methods

- If we can't represent a query with the help of a method name or criteria, we can do something more low level, use the @Query annotation.

```
@Query("{ 'name' : ?0 }")
List<User> findUsersByName(String name);
```

- This method should return users by name. The placeholder ?0 references the first parameter of the method.

```
List<User> users = userRepository.findUsersByName("Eric");
```

- In order to implement *lt* and *gt* query:

```
@Query("{ 'age' : { $gt: ?0, $lt: ?1 } }")
List<User> findUsersByAgeBetween(int ageGT, int ageLT);
```

- Now that the method has 2 parameters, we're referencing each of these by index in the raw query, ?0 and ?1:

```
List<User> users = userRepository.findUsersByAgeBetween(20, 50);
```

Read Operations - Criteria Query

- Queries can be created at runtime.
- In order to run them, auto wire MongoTemplate into any Spring component.
- *where, is, lt ,gt and sort* are some of the available criteria.

```
[
    {
        "_id" : ObjectId("55c0e5e5511f0a164a581907"),
        "_class" : "org.baeldung.model.User",
        "name" : "Eric",
        "age" : 45
    },
    {
        "_id" : ObjectId("55c0e5e5511f0a164a581908"),
        "_class" : "org.baeldung.model.User",
        "name" : "Antony",
        "age" : 55
    }
}
```

```
Query query = new Query();
query.addCriteria(Criteria.where("name").is("Eric"));
List<User> users = mongoTemplate.find(query, User.class);
```

```
Query query = new Query();
query.addCriteria(Criteria.where("age").lt(50).gt(20));
List<User> users = mongoTemplate.find(query,User.class);
```

```
Query query = new Query();
query.with(Sort.by(Sort.Direction.ASC, "age"));
List<User> users = mongoTemplate.find(query,User.class);
```

Asking for a page size of 2:

```
final Pageable pageableRequest = PageRequest.of(0, 2);
Query query = new Query();
query.with(pageableRequest);
```

Insert

- Insert statement creates the collection if it doesn’t exist with the new document:

First, we'll see the state of the database before running the *insert*:

```
{
}
```

Now we'll insert a new user:

```
User user = new User();
user.setName("Jon");
userRepository.insert(user);
```

And here's the end state of the database:

```
{
    "_id" : ObjectId("55b4fda5830b550a8c2ca25a"),
    "_class" : "com.baeldung.model.User",
    "name" : "Jon"
}
```

Save - Insert

- The save operation has save-or-update semantics: if an id is present, it performs an update, and if not, it does an insert:

When we now *save* a new user:

```
User user = new User();
user.setName("Albert");
mongoTemplate.save(user, "user");
```

the entity will be inserted in the database:

```
{
    "_id" : ObjectId("55b52bb7830b8c9b544b6ad5"),
    "_class" : "com.baeldung.model.User",
    "name" : "Albert"
}
```

Save - Update

Let's now look at *save* with update semantics, operating on an existing entity:

```
{
    "_id" : ObjectId("55b52bb7830b8c9b544b6ad5"),
    "_class" : "com.baeldung.model.User",
    "name" : "Jack"
}
```

When we *save* the existing user, we will update it:

```
user = mongoTemplate.findOne(
  Query.query(Criteria.where("name").is("Jack")), User.class);
user.setName("Jim");
mongoTemplate.save(user, "user");
```

The database will look like this:

```
{
    "_id" : ObjectId("55b52bb7830b8c9b544b6ad5"),
    "_class" : "com.baeldung.model.User",
    "name" : "Jim"
}
```

Delete

- First access any object with their fields and then call delete on repository:

Here's the state of the database before calling *delete*:

```
{
    "_id" : ObjectId("55b5ffa5511fee0e45ed614b"),
    "_class" : "com.baeldung.model.User",
    "name" : "Benn"
}
```

Let's run *delete*:

```
userRepository.delete(user);
```

And here's our result:

```
{
}
```

Official Documentation

- Spring Data MongoDB
  - https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/#preface
- MongoDB Spring Tutorial
  - https://www.mongodb.com/compatibility/spring-boot