Spring Boot and MongoDB

In this tutorial we'll use Spring Boot to access data from a MongoDB Atlas cluster. To follow along you'll need to sign in to MongoDB Atlas.

Sign up for Atlas

Sign in >

Java developers often make use of the Spring framework. Spring Boot allows developers to create microservices and web applications using Spring. By using this tool, you can rapidly create standalone applications without needing to make unnecessary configuration changes.

MongoDB is a great fit for Java developers who need a database. Combining Spring Boot and MongoDB results in applications that are fast, secure, reliable, and require minimal development time.

This tutorial shows how Spring Boot and MongoDB come together seamlessly with Spring Data MongoDB and will help you build a full Spring application.

Table of contents:

Spring Boot is an auto-configured microservice-based web framework that provides built-in features for security and database access.

With Spring boot, we can quickly create stand-alone applications without having to make too many configuration changes (as we will see later). MongoDB is the most popular NoSQL database because of the ease with which data can be stored and retrieved. Combining Spring Boot and MongoDB results in applications that are fast, secure, reliable, and require minimum development time.

This tutorial demonstrates how we can integrate Spring Boot with MongoDB using the Spring Data MongoDB API.

Getting started with MongoDB and Spring Boot

Spring is an application framework for Java, based on the MVC (Model-View-Controller) framework. Spring Boot is built on top of Spring and is mainly used for REST APIs. It has four layers:

1. The presentation layer, for front end
2. The business layer, for business logic and validation
3. The persistence layer, for translating business objects to database objects
4. The database layer, for handling CRUD operations

MongoDB can handle large amounts of structured and unstructured data, making it a database of choice for web applications. The Spring framework provides powerful connectors to easily perform database operations with MongoDB.

In this tutorial, we'll build a Spring Boot application, focusing on persistence and database layers. We'll run our program from our IDE to focus on CRUD operations. We'll add Spring Boot MongoDB configurations so that we can use Spring Boot with MongoDB.

What We Will Build

We're going to build a grocery list for a user. In this tutorial, we'll demonstrate the following:

1. Creating a POJO (Plain Old Java Object) to represent a grocery item
2. CRUD operations using MongoRepository
3. An alternate approach for document updates using MongoTemplate.

What We Need

We'll need:

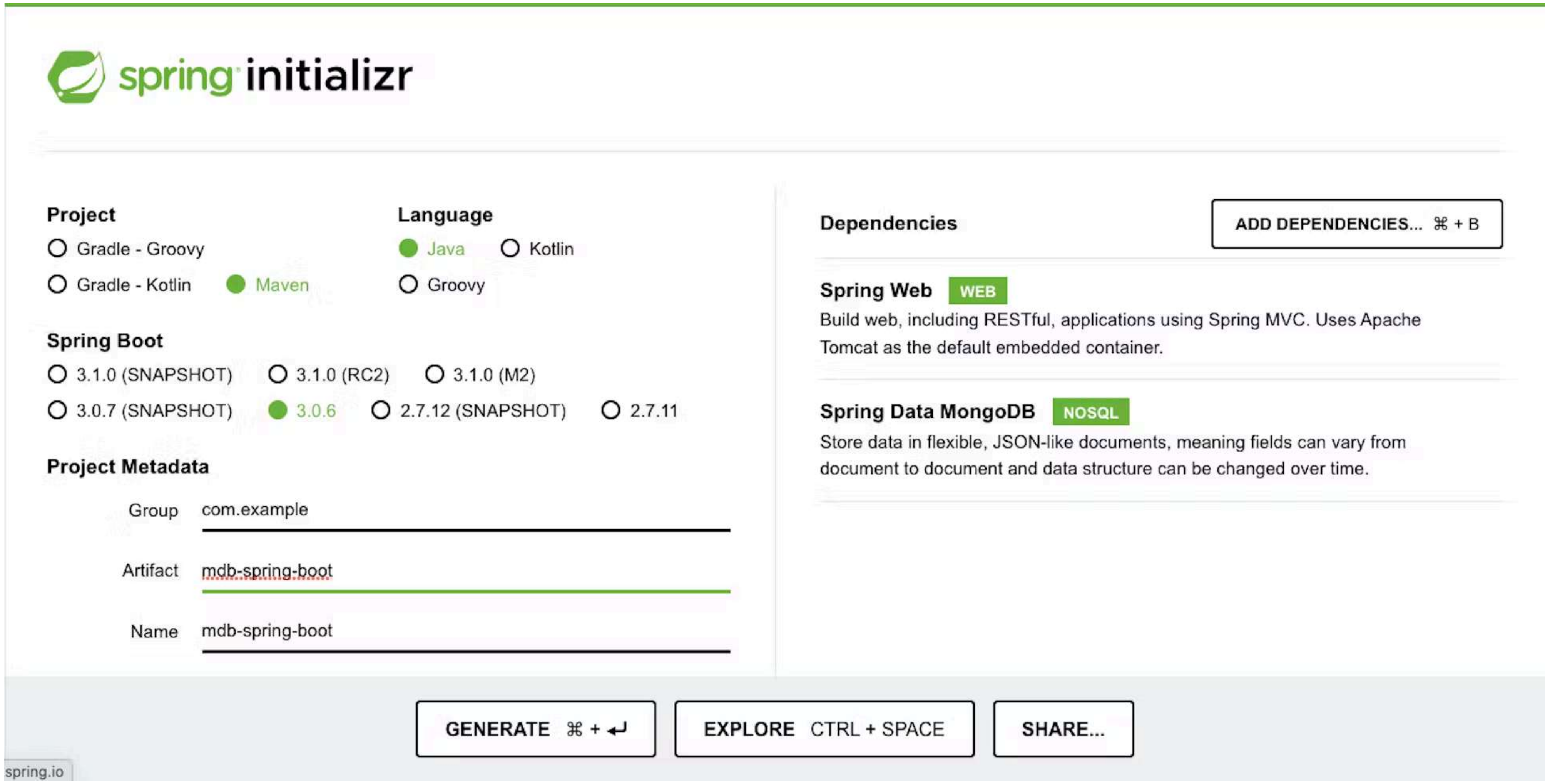
- A MongoDB Atlas cluster
- Java 8 or later
- Spring Initializr
- Maven
- Your IDE of choice

The complete code for this tutorial can be found on GitHub.

Getting started with Spring Initializr

Let's use Spring Initializr to generate a Spring Boot project. Using Spring Initializr takes care of creating a pom.xml file, which Maven uses for dependency management.

Select the following options:



spring initializr

Project

○ Gradle - Groovy

○ Gradle - Kotlin

● Maven

Language

● Java

○ Kotlin

○ Groovy

Spring Boot

○ 3.1.0 (SNAPSHOT)

○ 3.1.0 (RC2)

○ 3.1.0 (M2)

○ 3.0.7 (SNAPSHOT)

● 3.0.6

○ 2.7.12 (SNAPSHOT)

○ 2.7.11

Project Metadata

Group	com.example
Artifact	mdb-spring-boot
Name	mdb-spring-boot

Dependencies

ADD DEPENDENCIES... ⌘ + B

Spring Web WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Data MongoDB NOSQL

Store data in flexible, JSON-like documents, meaning fields can vary from document to document and data structure can be changed over time.

GENERATE ⌘ + ↵

EXPLORE CTRL + SPACE

SHARE...

spring.io

- Maven Project
- Java language

- Dependencies: Spring Web and Spring Data MongoDB

Enter the project metadata (as shown in the image above) and select the JAR option.

MongoDB model implementation

Our model is the POJO, or in this case, the GroceryItem class.

Let's create a package called com.example.mdbspringboot.model and add the class GroceryItem.java.

We use the annotation @Document to specify the collection name that will be used by the model. If the collection doesn't exist, MongoDB will create it.

```java

        private String name;
        private int quantity;

        this.id = id;
        this.name = name;
        this.quantity = quantity;
        this.category = category;
    }
}
```

If your IDE is Eclipse, you can use the Source -> Generate Getters and Setters option to create getters and setters for this code.

You'll note that in the above sample, the primary key in our MongoDB document is specified using the @Id annotation. If we don't do this, MongoDB will automatically generate an **_id** when creating the document.

Spring Boot MongoDB 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.

Let's create a package called com.example.mdbspringboot.repository to store all the repository files.

We first create an ItemRepository public interface, which extends the MongoRepository interface.

```java
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();

}
```

The first method, findItemByName, requires a parameter for the query, i.e., the field by which to filter the query. We specify this with the annotation @Query. The second method uses the category field to get all the items of a particular category. We only want to project the field's name and quantity in the query response, so we set those fields to 1. We reuse the method count() as it is.

MongoDB and Spring Boot CRUD examples

To connect to MongoDB Atlas, we specify the connection string in the application.properties file in the src/main/resources folder.

The connection string for a cluster can be found in the Atlas UI. There is no need to write connection-related code in any other file. Spring Boot takes care of the database connection for us.

```
spring.data.mongodb.uri=mongodb+srv://<username>:<pwd>@<cluster>.mongodb.net/mygrocerylist
spring.data.mongodb.database=mygrocerylist
```

We're also specifying the database name here. If it doesn't exist, MongoDB will create one.

In this Spring Boot MongoDB example, we are not using the Controller and the View. We will use a CommandLineRunner to view the output on the console.

Create the main application class MdbSpringBootApplication.java in the root package com.example.mdbspringboot:

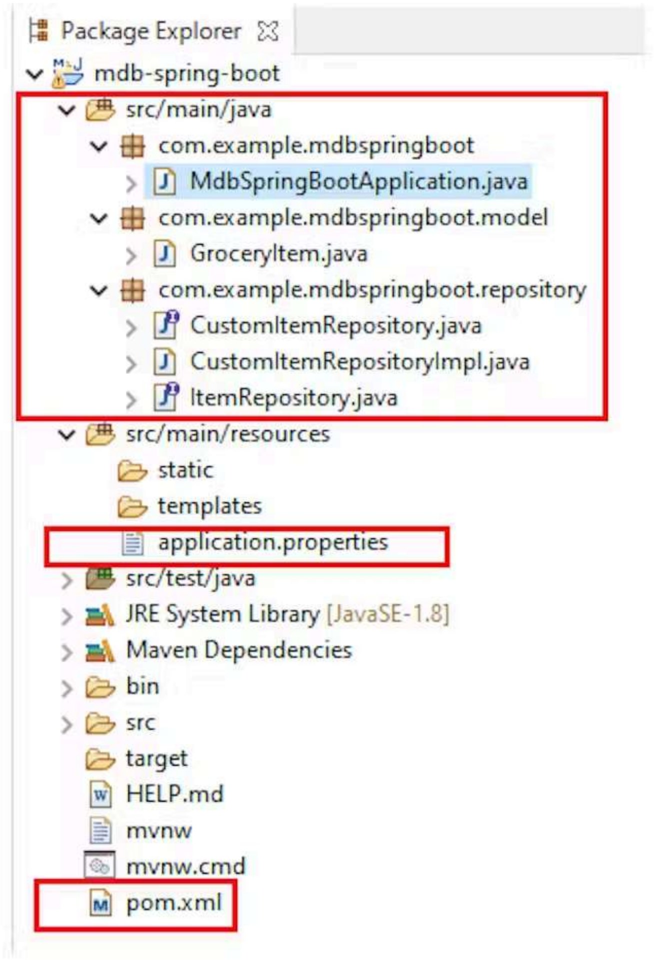
```java
@SpringBootApplication
@EnableMongoRepositories
public class MdbSpringBootApplication implements CommandLineRunner{

    @Autowired
    ItemRepository groceryItemRepo;

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

Our class MdbSpringBootApplication implements the CommandLineRunner interface to run the Spring application. ItemRepository is Autowired, allowing Spring to find it automatically.

Spring initializes the Application Context using the @SpringBootApplication annotation. We also activate the Mongo Repositories using @EnableMongoRepositories. Our project structure should be similar to the below structure now:



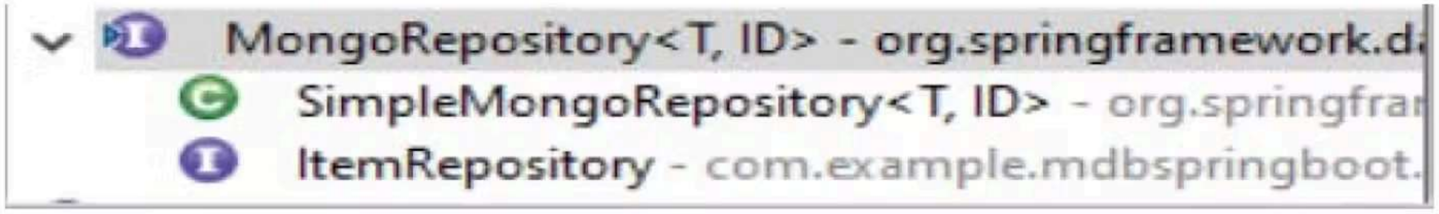
```
Package Explorer
mdb-spring-boot
  src/main/java
    com.example.mdbspringboot
      MdbSpringBootApplication.java
    com.example.mdbspringboot.model
      GroceryItem.java
    com.example.mdbspringboot.repository
      CustomItemRepository.java
      CustomItemRepositoryImpl.java
      ItemRepository.java
  src/main/resources
    static
    templates
    application.properties
  src/test/java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  bin
  src
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

Let’s now add the repository methods to the main class for CRUD operations:

Create operation using Spring Boot MongoDB

In order to create new documents, we will use the save method. The save method is available to us through the SimpleMongoRepository class, which implements the

MongoRepository interface. Our ItemRepository interface extends MongoRepository.



MongoRepository<T, ID> - org.springframework.d
SimpleMongoRepository<T, ID> - org.springfra
ItemRepository - com.example.mdbspringboot.

The save method will take a GroceryItem object as a parameter. Let's create five grocery items (documents) and save them using the save method in the following code:

```java
//CREATE
    void createGroceryItems() {
        System.out.println("Data creation started...");
        groceryItemRepo.save(new GroceryItem("Whole Wheat Biscuit", "Whole Wheat Biscuit", 5, "snacks"));
        groceryItemRepo.save(new GroceryItem("Kodo Millet", "XYZ Kodo Millet healthy", 2, "millets"));
        groceryItemRepo.save(new GroceryItem("Dried Red Chilli", "Dried Whole
Red Chilli", 2, "spices"));
        groceryItemRepo.save(new GroceryItem("Pearl Millet", "Healthy Pearl
Millet", 1, "millets"));
        groceryItemRepo.save(new GroceryItem("Cheese Crackers", "Bonny Cheese
Crackers Plain", 6, "snacks"));
        System.out.println("Data creation complete...");
```

Read operations using Spring Boot MongoDB

In this application, we perform four different read operations:

- Fetching all the documents using findAll()
- Getting a single document by name
- Getting a list of items by category
- Getting the count of items

```
// READ
    // 1. Show all the data
     public void showAllGroceryItems() {

         groceryItemRepo.findAll().forEach(item ->
System.out.println(getItemDetails(item)));
```

```java
    }

    // 2. Get item by name
    public void getGroceryItemByName(String name) {
        System.out.println("Getting item by name: " + name);
        GroceryItem item = groceryItemRepo.findItemByName(name);
        System.out.println(getItemDetails(item));
    }
```

We can create a helper method to display the output of read operations in a readable format:

```java
// Print details in readable form

    public String getItemDetails(GroceryItem item) {

        System.out.println(
        "Item Name: " + item.getName() +
        ", \nQuantity: " + item.getQuantity() +
        ", \nItem Category: " + item.getCategory()
        );

        return "";
    }
```

Updates using Spring Boot MongoDB

To change the category from "snacks" to "munchies," we first need to find all documents with the category "snacks," set their categories to "munchies," then save all the modified documents.

```
public void updateCategoryName(String category) {

        // Change to this new value
        String newCategory = "munchies";

        // Find all the items with the category snacks
        List<GroceryItem> list = groceryItemRepo.findAll(category);

        list.forEach(item -> {
            // Update the category in each document
            item.setCategory(newCategory);
        });

```

Deletes using Spring Boot MongoDB

To remove an item from our grocery list, we can delete it by ID using deleteById.

```
// DELETE
    public void deleteGroceryItem(String id) {
        groceryItemRepo.deleteById(id);
        System.out.println("Item with id " + id + " deleted...");
    }
```

To delete *all* the items, we can use the groceryItemRepo.deleteAll() method.

Putting it all together

Next, we implement the CommandLineRunner.run() method to call the above methods:

```
public void run(String... args) {

        System.out.println("-----CREATE GROCERY ITEMS-----\n");

        createGroceryItems();

        System.out.println("\n-----SHOW ALL GROCERY ITEMS-----\n");

        showAllGroceryItems();

        System.out.println("\n-----GET ITEM BY NAME-----\n");

        getGroceryItemByName("Whole Wheat Biscuit");

```

The output should be similar to the following:

```
-----CREATE GROCERY ITEMS-----

Data creation started...
Data creation complete...

-----SHOW ALL GROCERY ITEMS-----

Item Name: Whole Wheat Biscuit,
Item Quantity: 5,
Item Category: snacks

Item Name: XYZ Kodo Millet healthy,
Item Quantity: 2,
Item Category: millets
```

Using MongoTemplate

To perform update operations using a particular field, we can also use the MongoTemplate class. The nice thing about MongoTemplate is that the update can be done in a single database interaction.

To use MongoTemplate, we create a custom repository where we build the update query.

Let's write a method to update the quantity of a grocery item.

Create an interface CustomItemRepository:

```java
public interface CustomItemRepository {

    void updateItemQuantity(String name, float newQuantity);

}
```

We can add as many methods as we need and provide the implementations in the CustomItemRepositoryImpl class:

```java
@Component
public class CustomItemRepositoryImpl implements CustomItemRepository {

    @Autowired
    MongoTemplate mongoTemplate;

    public void updateItemQuantity(String name, float newQuantity) {
        Query query = new Query(Criteria.where("name").is(name));
        Update update = new Update();
        update.set("quantity", newQuantity);

        UpdateResult result = mongoTemplate.updateFirst(query, update,
GroceryItem.class);

```

Since MongoTemplate is @Autowired, Spring will inject the object dependency. The @Component annotation will allow Spring itself to detect the CustomItemRepository interface.

The next step is to call this method from our main class. We'll declare our customRepo similar to how we declared the groceryItemRepo:

```
    @Autowired
CustomItemRepository customRepo;
```

And we'll need a method in our main class to call the customRepo method.

```java
// UPDATE
        public void updateItemQuantity(String name, float newQuantity) {
            System.out.println("Updating quantity for " + name);
            customRepo.updateItemQuantity(name, newQuantity);
        }
```

Add the above method in the run method to call it when the application is executed:

```java
System.out.println("\n-----UPDATE QUANTITY OF A GROCERY ITEM-----\n");
```

```

        updateItemQuantity("Bonny Cheese Crackers Plain", 10);
```

The resulting output should be:

```
-----UPDATE QUANTITY OF A GROCERY ITEM-----

Updating quantity for Bonny Cheese Crackers Plain
1 document(s) updated..
```

In the MongoRepository example mentioned earlier, we had to do three operations (find, set, save). In this case, we updated in a single database transaction!

It's easy to connect MongoDB Atlas with Spring Boot

In this article, we've gone over the basic concepts of using Spring Boot with MongoDB and built a full Spring Boot application. To go beyond this Spring Boot starter, or to learn more about the core capabilities of Spring Data, refer to our handy guide.

Try this tutorial for yourself.

Sign up for Atlas

FAQ

How does MongoDB connect to Spring Boot? +

What is Spring Boot used for?

+

Where should I go from here?

+

English