ALL GUIDES

Accessing Data with MongoDB

Get the Code
Go To Repo

This guide walks you through the process of using Spring Data MongoDB to build an application that stores data in and retrieves it from MongoDB, a document-based database.

What You Will build

You will store `Customer` POJOs (Plain Old Java Objects) in a MongoDB database by using Spring Data MongoDB.

What You Need

- About 15 minutes
- A favorite text editor or IDE
- Java 17 or later
- Gradle 7.5+ or Maven 3.5+
- You can also import the code straight into your IDE:
  - Spring Tool Suite (STS)
  - IntelliJ IDEA
  - VSCode

How to complete this guide

Like most Spring Getting Started guides, you can start from scratch and complete each step or you can bypass basic setup steps that are already familiar to you. Either way, you end up with working code.

To **start from scratch**, move on to Starting with Spring Initializr.

To **skip the basics**, do the following:

- Download and unzip the source repository for this guide, or clone it using Git: `git clone https://github.com/spring-guides/gs-accessing-data-mongodb.git`
- cd into `gs-accessing-data-mongodb/initial`
- Jump ahead to Install and Launch MongoDB.

When you finish, you can check your results against the code in `gs-accessing-data-mongodb/complete`.

Starting with Spring Initializr

You can use this pre-initialized project and click Generate to download a ZIP file. This project is configured to fit the examples in this tutorial.

To manually initialize the project:

1. Navigate to https://start.spring.io. This service pulls in all the dependencies you need for an application and does most of the setup for you.
2. Choose either Gradle or Maven and the language you want to use. This guide assumes that you chose Java.
3. Click **Dependencies** and select **Spring Data MongoDB**.
4. Click **Generate**.
5. Download the resulting ZIP file, which is an archive of a web application that is configured with your choices.

> If your IDE has the Spring Initializr integration, you can complete this process from your IDE.

> You can also fork the project from Github and open it in your IDE or other editor.

Install and Launch MongoDB

With your project set up, you can install and launch the MongoDB database.

If you use a Mac with Homebrew, you can run the following command:

```
$ brew install mongodb
```

With MacPorts, you can run the following command:

```
$ port install mongodb
```

For other systems with package management, such as Redhat, Ubuntu, Debian, CentOS, and Windows, see the instructions at https://docs.mongodb.org/manual/installation/.

After you install MongoDB, you can launch it in a console window by running the following command (which also starts up a server process):

```
$ mongod
```

You should see output similar to the following:

```
all output going to: /usr/local/var/log/mongodb/mongo.log
```

Define a Simple Entity

MongoDB is a NoSQL document store. In this example, you store `Customer` objects. The following listing shows the Customer class (in `src/main/java/com/example/accessingdatamongodb/Customer.java`):

COPY

```
package com.example.accessingdatamongodb;

import org.springframework.data.annotation.Id;


public class Customer {

  @Id
  public String id;

  public String firstName;
  public String lastName;

  public Customer() {}

  public Customer(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  @Override
  public String toString() {
    return String.format(
        "Customer[id=%s, firstName='%s', lastName='%s']",
        id, firstName, lastName);
  }

}
```

Here you have a `Customer` class with three attributes: `id`, `firstName`, and `lastName`. The `id` is mostly for internal use by MongoDB. You also have a single constructor to populate the entities when creating a new instance.

> In this guide, the typical getters and setters have been left out for brevity.

`id` fits the standard name for a MongoDB ID, so it does not require any special annotation to tag it for Spring Data MongoDB.

The other two properties, `firstName` and `lastName`, are left unannotated. It is assumed that they are mapped to fields that share the same name as the properties themselves.

The convenient `toString()` method prints out the details about a customer.

> MongoDB stores data in collections. Spring Data MongoDB maps the `Customer` class into a collection called `customer`. If you want to change the name of the collection, you can use Spring Data MongoDB's `@Document` annotation on the class.

Create Simple Queries

Spring Data MongoDB focuses on storing data in MongoDB. It also inherits functionality from the Spring Data Commons project, such as the ability to derive queries. Essentially, you need not learn the query language of MongoDB. You can write a handful of methods and the queries are written for you.

To see how this works, create a repository interface that queries `Customer` documents, as the following listing (in `src/main/java/com/example/accessingdatamongodb/CustomerRepository.java`) shows:

COPY

```
package com.example.accessingdatamongodb;

import java.util.List;

import org.springframework.data.mongodb.repository.MongoRepository;

public interface CustomerRepository extends MongoRepository<Customer, String> {

  public Customer findByFirstName(String firstName);
  public List<Customer> findByLastName(String lastName);

}
```

`CustomerRepository` extends the `MongoRepository` interface and plugs in the type of values and ID that it works with: `Customer` and `String`, respectively. This interface comes with many operations, including standard CRUD operations (create, read, update, and delete).

You can define other queries by declaring their method signatures. In this case, add `findByFirstName`, which essentially seeks documents of type `Customer` and finds the documents that match on `firstName`.

You also have `findByLastName`, which finds a list of people by last name.

In a typical Java application, you write a class that implements `CustomerRepository` and craft the queries yourself. What makes Spring Data MongoDB so useful is the fact that you need not create this implementation. Spring Data MongoDB creates it on the fly when you run the application.

Now you can wire up this application and see what it looks like!

Create an Application Class

Spring Initializr creates a simple class for the application. The following listing shows the class that Initializr created for this example (in `src/main/java/com/example/accessingdatamongodb/AccessingDataMongodbApplication.java`):

COPY

```
package com.example.accessingdatamongodb;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class AccessingDataMongodbApplication {

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

}
```

`@SpringBootApplication` is a convenience annotation that adds all of the following:

- `@Configuration`: Tags the class as a source of bean definitions for the application context.
- `@EnableAutoConfiguration`: Tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings. For example, if `spring-webmvc` is on the classpath, this annotation flags the application as a web application and activates key behaviors, such as setting up a `DispatcherServlet`.
- `@ComponentScan`: Tells Spring to look for other components, configurations, and services in the `com/example` package, letting it find the controllers.

The `main()` method uses Spring Boot's `SpringApplication.run()` method to launch an application. Did you notice that there was not a single line of XML? There is no `web.xml` file, either. This web application is 100% pure Java and you did not have to deal with configuring any plumbing or infrastructure.

Spring Boot automatically handles those repositories as long as they are included in the same package (or a sub-package) of your `@SpringBootApplication` class. For more control over the registration process, you can use the `@EnableMongoRepositories` annotation.

> By default, `@EnableMongoRepositories` scans the current package for any interfaces that extend one of Spring Data's repository interfaces. You can use its `basePackageClasses=MyRepository.class` to safely tell Spring Data MongoDB to scan a different root package by type if your project layout has multiple projects and it does not find your repositories.

Spring Data MongoDB uses the `MongoTemplate` to execute the queries behind your `find*` methods. You can use the template yourself for more complex queries, but this guide does not cover that. (see the Spring Data MongoDB Reference Guide)

Now you need to modify the simple class that the Initializr created for you. You need to set up some data and use it to generate output. The following listing shows the finished `AccessingDataMongodbApplication` class (in `src/main/java/com/example/accessingdatamongodb/AccessingDataMongodbApplication.java`):

COPY

```
package com.example.accessingdatamongodb;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class AccessingDataMongodbApplication implements CommandLineRunner {

  @Autowired
  private CustomerRepository repository;

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

  @Override
  public void run(String... args) throws Exception {

    repository.deleteAll();

    // save a couple of customers
    repository.save(new Customer("Alice", "Smith"));
    repository.save(new Customer("Bob", "Smith"));

    // fetch all customers
    System.out.println("Customers found with findAll():");
    System.out.println("-------------------------------");
    for (Customer customer : repository.findAll()) {
      System.out.println(customer);
    }
    System.out.println();

    // fetch an individual customer
    System.out.println("Customer found with findByFirstName('Alice'):");
    System.out.println("--------------------------------");
    System.out.println(repository.findByFirstName("Alice"));

    System.out.println("Customers found with findByLastName('Smith'):");
    System.out.println("--------------------------------");
    for (Customer customer : repository.findByLastName("Smith")) {
      System.out.println(customer);
    }

  }

}
```

`AccessingDataMongodbApplication` includes a `main()` method that autowires an instance of `CustomerRepository`. Spring Data MongoDB dynamically creates a proxy and injects it there. We use the `CustomerRepository` through a few tests. First, it saves a handful of `Customer` objects, demonstrating the `save()` method and setting up some data to use. Next, it calls `findAll()` to fetch all `Customer` objects from the database. Then it calls `findByFirstName()` to fetch a single `Customer` by her first name. Finally, it calls `findByLastName()` to find all customers whose last name is `Smith`.

> By default, Spring Boot tries to connect to a locally hosted instance of MongoDB. Read the reference docs for details on pointing your application to an instance of MongoDB hosted elsewhere.

Build an executable JAR

You can run the application from the command line with Gradle or Maven. You can also build a single executable JAR file that contains all the necessary dependencies, classes, and resources and run that. Building an executable jar makes it easy to ship, version, and deploy the service as an application throughout the development lifecycle, across different environments, and so forth.

If you use Gradle, you can run the application by using `./gradlew bootRun`. Alternatively, you can build the JAR file by using `./gradlew build` and then run the JAR file, as follows:

```
java -jar build/libs/gs-accessing-data-mongodb-0.1.0.jar
```

If you use Maven, you can run the application by using `./mvnw spring-boot:run`. Alternatively, you can build the JAR file with `./mvnw clean package` and then run the JAR file, as follows:

```
java -jar target/gs-accessing-data-mongodb-0.1.0.jar
```

> The steps described here create a runnable JAR. You can also build a classic WAR file.

As `AccessingDataMongodbApplication` implements `CommandLineRunner`, the `run` method is automatically invoked when Spring Boot starts. You should see something like the following (with other output, such as queries, as well):

```
== Customers found with findAll():
Customer[id=51df1b0a3004cb49c50210f8, firstName='Alice', lastName='Smith']
Customer[id=51df1b0a3004cb49c50210f9, firstName='Bob', lastName='Smith']

== Customer found with findByFirstName('Alice'):
Customer[id=51df1b0a3004cb49c50210f8, firstName='Alice', lastName='Smith']
== Customers found with findByLastName('Smith'):
Customer[id=51df1b0a3004cb49c50210f8, firstName='Alice', lastName='Smith']
Customer[id=51df1b0a3004cb49c50210f9, firstName='Bob', lastName='Smith']
```

Summary

Congratulations! You set up a MongoDB server and wrote a simple application that uses Spring Data MongoDB to save objects to and fetch them from a database, all without writing a concrete repository implementation.

> If you want to expose MongoDB repositories with a hypermedia-based RESTful front end with little effort, read Accessing MongoDB Data with REST.

See Also

The following guides may also be helpful:

- Accessing MongoDB Data with REST
- Accessing Data with JPA
- Accessing Data with Gemfire
- Accessing data with MySQL
- Accessing Data with Neo4j

Want to write a new guide or contribute to an existing one? Check out our contribution guidelines.