Spring Data MongoDB – Configure Connection

Last updated: May 11, 2024

Written by: Ulisses Lima

Reviewed by: Luis Javier Peris

Spring Data

MongoDB | Spring Data MongoDB

1. Overview

In this tutorial, **we'll learn different ways to configure a MongoDB connection in a Spring Boot application**. We'll use the powerful capabilities offered by the Spring Data MongoDB project. By leveraging the Spring Data MongoDB project, we gain access to a rich set of tools and functionalities that streamline the process of working with MongoDB databases in a Spring environment.

By delving into Spring's flexible configuration options, we'll explore various approaches for establishing database connections. Through hands-on examples, we'll create separate applications for each approach, enabling us to select the most appropriate configuration method tailored to our specific requirements.

2. Testing Our Connections

Before we start building our applications, we'll create a test class. Let's start with a few constants we'll reuse:

```
public class MongoConnectionApplicationLiveTest {
    private static final String HOST = "localhost";
    private static final String PORT = "27017";
    private static final String DB = "baeldung";
    private static final String USER = "admin";
    private static final String PASS = "password";

    // test cases
}
```

Our tests consist of running our application, and then trying to insert a document in a collection called *"items"*. After inserting our document, we should receive an *"_id"* from our database, and we'll consider the test successful. Now let's create a helper method for that:

```
private void assertInsertSucceeds(ConfigurableApplicationContext context) {
    String name = "A";

    MongoTemplate mongo = context.getBean(MongoTemplate.class);
    Document doc = Document.parse("{\"name\":\"" + name + "\"}");
    Document inserted = mongo.insert(doc, "items");

    assertNotNull(inserted.get("_id"));
    assertEquals(inserted.get("name"), name);
}
```

Our method receives the Spring context from our application so that we can retrieve the *MongoTemplate* instance. Next, we'll build a simple JSON document from a string with *Document.parse()*.

This way, we don't need to create a repository or a document class. Then, after inserting, we'll assert the properties in our inserted document are what we expect.

It's important to note that we need to run a real MongoDB instance. For this, we can run MongoDB as a docker container.

3. Configuring Connections via Properties

To configure MongoDB connections in our Spring Boot application, we typically use properties. In properties, we define essential connection details such as the database host, port, authentication credentials, and database name. We'll see these properties in detail in the following subsections.

3.1. Using the *application.properties*

Our first example is the most common way of configuring connections. **We just have to provide our database information in our *application.properties*:**

```
spring.data.mongodb.host=localhost
spring.data.mongodb.port=27017
spring.data.mongodb.database=baeldung
spring.data.mongodb.username=admin
spring.data.mongodb.password=password
```

All available properties reside in the *MongoProperties* class from Spring Boot. We can also use this class to check default values. **We can define any configuration in our properties file via application arguments.**

In our application class, we need to exclude the *EmbeddedMongoAutoConfiguration* class to get up and running:

```
@SpringBootApplication(exclude={EmbeddedMongoAutoConfiguration.class})
public class SpringMongoConnectionViaPropertiesApp {

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

This configuration is all we need to connect to our database instance. The *@SpringBootApplication* annotation includes *@EnableAutoConfiguration*. It takes care of discovering that our application is a MongoDB application based on our classpath.

To test it, we can use *SpringApplicationBuilder* to get a reference to the application context. **Then, to assert our connection is valid, we'll use the *assertInsertSucceeds* method created earlier:**

```
@Test
public void whenPropertiesConfig_thenInsertSucceeds() {
    SpringApplicationBuilder app = new SpringApplicationBuilder(SpringMongoConnectionViaPropertiesApp.class);
    app.run();

    assertInsertSucceeds(app.context());
}
```

In the end, our application was successfully connected using our *application.properties* file.

3.2. Overriding Properties With Command Line Arguments

We can override our properties file when running our application with command line arguments. These are passed to the application when run with the java command, mvn command, or IDE configuration. The method to provide these will depend on the command we're using.

Let's see an example using *mvn* to run our Spring Boot application:

```
mvn spring-boot:run -Dspring-boot.run.arguments='--spring.data.mongodb.port=7017 --
spring.data.mongodb.host=localhost'
```

To use it, we specify our properties as values to the *spring-boot.run.arguments* argument. We use the same property names but prefix them with two dashes. **Since Spring Boot 2, multiple properties should be separated by a space.** Finally, after running the command, there shouldn't be any errors.

Options configured this way always take precedence over the properties file. **This option is useful when we need to change our application parameters without changing our properties file.** For instance, if our credentials have changed and we can't connect anymore.

To simulate this in our tests, we can set system properties before running our application. We can also override our *application.properties* with the *properties* method:

```
@Test
public void givenPrecedence_whenSystemConfig_thenInsertSucceeds() {
    System.setProperty("spring.data.mongodb.host", HOST);
    System.setProperty("spring.data.mongodb.port", PORT);
    System.setProperty("spring.data.mongodb.database", DB);
    System.setProperty("spring.data.mongodb.username", USER);
    System.setProperty("spring.data.mongodb.password", PASS);

    SpringApplicationBuilder app = new SpringApplicationBuilder(SpringMongoConnectionViaPropertiesApp.class)
      .properties(
        "spring.data.mongodb.host=oldValue",
        "spring.data.mongodb.port=oldValue",
        "spring.data.mongodb.database=oldValue",
        "spring.data.mongodb.username=oldValue",
        "spring.data.mongodb.password=oldValue"
      );
    app.run();

    assertInsertSucceeds(app.context());
}
```

As a result, the old values in our properties file won't affect our application because system properties have more precedence. This can be useful when we need to restart our application with new connection details without changing the code.

3.3. Using the Connection URI Property

It's also possible to use a single property instead of the individual host, port, etc.:

```
spring.data.mongodb.uri="mongodb://admin:password@localhost:27017/baeldung"
```

This property includes all values from the initial properties, so we don't need to specify all five. Let's check the basic format:

```
mongodb://<username>:<password>@<host>:<port>/<database>
```

The *database* part in the URI is, more specifically, the default auth DB. **Most importantly, the *spring.data.mongodb.uri* property can't be specified along with the individual ones for host, port, and credentials.** Otherwise, we'll get the following error when running our application:

```
@Test
public void givenConnectionUri_whenAlsoIncludingIndividualParameters_thenInvalidConfig() {
    System.setProperty(
      "spring.data.mongodb.uri",
      "mongodb://" + USER + ":" + PASS + "@" + HOST + ":" + PORT + "/" + DB
    );

    SpringApplicationBuilder app = new SpringApplicationBuilder(SpringMongoConnectionViaPropertiesApp.class)
      .properties(
        "spring.data.mongodb.host=" + HOST,
        "spring.data.mongodb.port=" + PORT,
        "spring.data.mongodb.username=" + USER,
        "spring.data.mongodb.password=" + PASS
      );

    BeanCreationException e = assertThrows(BeanCreationException.class, () -> {
        app.run();
    });

    Throwable rootCause = e.getRootCause();
    assertTrue(rootCause instanceof IllegalStateException);
    assertThat(rootCause.getMessage()
      .contains("Invalid mongo configuration, either uri or host/port/credentials/replicaSet must be
specified"));
}
```

In the end, this configuration option is not only shorter but sometimes required. That's because some options are only available through the connection string, like using *mongodb+srv* to connect to a replica set. As such, we'll only use this simpler configuration property for the next examples.

4. Java Setup With MongoClient

***MongoClient* represents our connection to a MongoDB database and is always created under the hood, but we can also set it up programmatically.** Despite being more verbose, this approach has a few advantages. Let's take a look at them over the next few subsections.

4.1. Connecting via *AbstractMongoClientConfiguration*

In our first example, we'll extend the *AbstractMongoClientConfiguration* class from Spring Data MongoDB in our application class:

```
@SpringBootApplication
public class SpringMongoConnectionViaClientApp extends AbstractMongoClientConfiguration {
    // main method
}
```

Next, we'll inject the properties we need:

```
@Value("${spring.data.mongodb.uri}")
private String uri;

@Value("${spring.data.mongodb.database}")
private String db;
```

To clarify, these properties could be hard-coded. Also, they could use names that differ from the expected Spring Data variables. **Most importantly, this time we're using a URI instead of individual connection properties, which can't be mixed.** Consequently, we can't reuse our *application.properties* for this application, and we should move it elsewhere.

AbstractMongoClientConfiguration requires us to override *getDatabaseName()*. This is because a database name isn't required in a URI:

```
protected String getDatabaseName() {
    return db;
}
```

At this point, because we're using default Spring Data variables, we'd already be able to connect to our database. Also, MongoDB creates the database if it doesn't exist. Let's test it:

```
@Test
public void whenClientConfig_thenInsertSucceeds() {
    SpringApplicationBuilder app = new SpringApplicationBuilder(SpringMongoConnectionViaClientApp.class);
    app.web(WebApplicationType.NONE)
      .run(
        "--spring.data.mongodb.uri=mongodb://" + USER + ":" + PASS + "@" + HOST + ":" + PORT + "/" + DB,
        "--spring.data.mongodb.database=" + DB
      );

    assertInsertSucceeds(app.context());
}
```

Finally, we can override *mongoClient()* to get an advantage over conventional configuration. **This method will use our URI variable to build a MongoDB client. That way, we can have a direct reference to it.** For instance, this enables us to list all the databases available from our connection:

```
@Override
public MongoClient mongoClient() {
    MongoClient client = MongoClients.create(uri);
    ListDatabasesIterable<Document> databases = client.listDatabases();
    databases.forEach(System.out::println);
    return client;
}
```

Configuring connections this way is useful if we want complete control over the MongoDB client's creation.

4.2. Creating a Custom *MongoClientFactoryBean*

In our next example, we'll create a *MongoClientFactoryBean*. **This time, we'll create a property called *custom.uri* to hold our connection configuration:**

```
@SpringBootApplication
public class SpringMongoConnectionViaFactoryApp {

    // main method

    @Bean
    public MongoClientFactoryBean mongo(@Value("${custom.uri}") String uri) {
        MongoClientFactoryBean mongo = new MongoClientFactoryBean();
        ConnectionString conn = new ConnectionString(uri);
        mongo.setConnectionString(conn);

        MongoClient client = mongo.getObject();
        client.listDatabaseNames()
          .forEach(System.out::println);
        return mongo;
    }
}
```

With this approach, we don't need to extend *AbstractMongoClientConfiguration*. We also have control over our *MongoClient*'s creation. For instance, by calling *mongo.setSingleton(false)*, we get a new client every time we call *mongo.getObject()*, instead of a singleton.

4.3. Set Connection Details With MongoClientSettingsBuilderCustomizer

In our last example, we're going to use a *MongoClientSettingsBuilderCustomizer*:

```
@SpringBootApplication
public class SpringMongoConnectionViaBuilderApp {

    // main method

    @Bean
    public MongoClientSettingsBuilderCustomizer customizer(@Value("${custom.uri}") String uri) {
        ConnectionString connection = new ConnectionString(uri);
        return settings -> settings.applyConnectionString(connection);
    }
}
```

We use this class to customize parts of our connection but still have auto-configuration for the rest. This is **helpful when we need to set just a few properties programmatically.**

5. Conclusion

In this article, we examined the different tools brought by Spring Data MongoDB. We used them to create connections in different ways. Moreover, we built test cases to guarantee our configurations worked as intended. Finally, we saw how configuration precedence could affect our connection properties.

As always, the source code is available over on GitHub.