A picture containing tableware, dishware

Description automatically generated



Mongo DB & GemFire

Brandon Rocha Díaz.

Software developer.

Shape, rectangle

Description automatically generated

**MONGODB**

MongoDB is document based NoSQL Database. Most widely used NoSQL Database in the world.

* Stores Data in JSON like documents. (BSON)
* Has very flexible structure to store the data
* Large amount of data can be easily managed

*SQL VS NOSQL*

|  |  |
| --- | --- |
| SQL | NoSQL |
| Relational database management system (rdbms) | Non-relational or distributed database system. |
| These databases have fixed or static or predefined schema | They have dynamic schema |
| These databases are not suited for hierarchical data storage. | These databases are best suited for hierarchical data storage. |
| These databases are best suited for complex queries | These databases are not so good for complex queries |
| Vertically Scalable | Horizontally scalable |
| Follows ACID property | Follows CAP(consistency, availability, partition tolerance) |
| Examples: MySQL, PostgreSQL, Oracle, MS-SQL Server etc | **Examples:**MongoDB, GraphQL, HBase, Neo4j, Cassandra etc |

*MONGODB TERMS*

|  |  |
| --- | --- |
| SQL | MONGO DB |
| Database | Database |
| Table | Collection |
| Column | Field |
| Row | Document |
| Foreign key | Sub Document (Reference) |

*WHEN TO USE SQL*

SQL is a good choice when working with related data. Relational databases are efficient, flexible, and easily accessed by any application. A benefit of a relational database is that when one user updates a specific record, every instance of the database automatically refreshes, and that information is provided in real-time.

SQL and a relational database make it easy to handle a great deal of information, scale as necessary and allow flexible access to data — only needing to update data once instead of changing multiple files, for instance. It’s also best for assessing data integrity. Since each piece of information is stored in a single place, there’s no problem with former versions confusing the picture.

Most of the big tech companies use SQL, which build their own database systems, use SQL to query and analyze data.

*WHEN TO USE NOSQL*

While SQL is valued for ensuring data validity, NoSQL is good when it’s more important that the availability of big data is fast. It’s also a good choice when a company will need to scale because of changing requirements.

NoSQL is also a good choice when there are large amounts of (or ever-changing) data sets or when working with flexible data models or needs that don't fit into a relational model. When working with large amounts of unstructured data, document databases are a good fit. For quick access to a key-value store without strong integrity guarantees, Redis may be the best choice. When a complex or flexible search across a lot of data is needed, Elastic Search is a good choice.

Scalability is a significant benefit of NoSQL databases. Unlike with SQL, their built-in sharding and high availability requirements allow horizontal scaling. Furthermore, NoSQL databases like Cassandra, developed by Facebook, handle massive amounts of data spread across many servers, having no single points of failure and providing maximum availability.

Other big companies that use NoSQL systems because they are dependent on large volumes of data not suited to a relational. In general, the more extensive the dataset, the more likely that NoSQL is a better choice.

**The *MongoTemplate***follows the standard template pattern in Spring and provides a ready-to-go, basic API to the underlying persistence engine.

**The repository** follows the Spring Data-centric approach and comes with more flexible and complex API operations, based on the well-known access patterns in all Spring Data projects.

For both, we need to start by defining the dependency — for example, in the *pom.xml*, with Maven:

<dependency>

<groupId>org.springframework.data</groupId>

<artifactId>spring-data-mongodb</artifactId>

<version>3.0.3.RELEASE</version>

</dependency>

*XML CONFIGURATION FOR MONGO TEMPLATE*

A simple configuration for mongo template:

<mongo:mongo-client id="mongoClient" host="localhost" />

<mongo:db-factory id="mongoDbFactory" dbname="test" mongo-client-ref="mongoClient" />

We first need to define the factory bean responsible for creating Mongo instances.

Then is needed to actually define (and configure) the template bean:

<bean id="mongoTemplate" class="org.springframework.data.mongodb.core.MongoTemplate"> <constructor-arg ref="mongoDbFactory"/> </bean>

And finally, we need to define a post processor to translate any MongoExceptions thrown in @Repository annotated classes:

<bean class= "org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"/>

*JAVA CONFIGURATION*

Let’s create a similar configuration using Java config by extending the base class for MongoDB configuration AbstractMongoConfiguration

@Configuration

public class MongoConfig extends AbstractMongoClientConfiguration {

@Override

protected String getDatabaseName() {

return "test"; }

@Override

public MongoClient mongoClient() {

ConnectionString connectionString = new ConnectionString("mongodb://localhost:27017/test"); MongoClientSettings mongoClientSettings = MongoClientSettings.builder() .applyConnectionString(connectionString) .build(); return MongoClients.create(mongoClientSettings);

}

@Override

public Collection getMappingBasePackages() { return Collections.singleton("com.baeldung"); } }

Note that we didn’t need to define MongoTemplate bean in the previous configuration since it’s already defined in AbstractMongoClientCondfiguration.

We also use our configuration from scratch without extending AbstractMongoClientConfiguration:

@Configuration

public class SimpleMongoConfig {

@Bean

public MongoClient mongo() {

ConnectionString connectionString = newConnectionString("mongodb://localhost:27017/test"); MongoClientSettings mongoClientSettings = MongoClientSettings.builder() .applyConnectionString(connectionString) .build();

return MongoClients.create(mongoClientSettings);

}

@Bean

public MongoTemplate mongoTemplate() throws Exception { return new MongoTemplate(mongo(), "test");

} }

*XML CONFIGURATION FOR MONGO REPOSITORY*

To make use of custom repositories (extending the MongoRepository), we need to continue the configuration from section and set up the repositories:

<mongo:repositories base-package="com.baeldung.repository" mongo-template-ref="mongoTemplate"/>

*JAVA CONFIGURATION*

Similarly, we’ll build on the configuration we already created, and add a new annotation into the mix:

@EnableMongoRepositories(basePackages = “com.baeldung.repository”)

*CREATE THE REPOSITORY*

After the configuration, we need to create a repository – extending the existing MongoRepository interface:

public interface UserRepository extends MongoRepository<User, String> { // }

Now we can auto-wire this UserRepository and use operations from MongoRepository or add custom operations.

*ANNOTATIONS*

Finally, let’s also go over the simple annotations that spring data uses to drive these API operations.

The field level @Id annotation can decorate any type, including long and string:

@Id

private String id;

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).

We'll next look at *@Document*:

@Document

public class User { // }

This annotation simply marks a class as being a domain object that needs to be persisted to the database, along with allowing us to choose the name of the collection to be used.

**GEMFIRE**

GemFire is a high-performance distributed data management infrastructure that sits between application cluster and back end data sources.

With GemFire, data can be managed in-memory, which makes the access faster. Spring Data provides an easy configuration and access to GemFire from Spring application.

To make use of the Spring Data GemFire support, we first need to add the following dependency in the pom.xml:

<dependency>

<groupId>org.springframework.data</groupId>

<artifactId>spring-data-gemfire</artifactId>

<version>1.9.1.RELEASE</version>

</dependency>

*GEMFIRE BASICS FEATURES*

The cache GemFire provides the essential data management services as well as manages the connectivity to other peers.

The cache configuration (cache.xml) describes how data will be distributed among different nodes:

<cache>

<region name="region"><region-attributes>

<cache-listener>

<class-name> ... </class-name>

</cache-listener> </region-attributes>

</region> ...

</cache>

*REGIONS*

Data regions are a logical grouping within a cache for a single data set.

Simply put, a region lets us store data in multiple VMs in the system without consideration to which node the data is stored within the cluster.

Regions are classified into three broad categories:

* **Replicated region** holds the complete set of data on each node. It gives a high read performance. Write operations are slower as the data update need to be propagated to each node:

<region name="myRegion" refid="REPLICATE"/>

* **Partitioned region** distributes the data so that each node only stores a part of region contents. A copy of the data is stored on one of the other nodes. It provides a good write performance.

<region name="myRegion" refid="PARTITION"/>

* **Local region** resides on the defining member node. There is no connectivity with other nodes within the cluster.

<region name="myRegion" refid="LOCAL"/>

GemFire provides a query language called OQL (Object Query Language) that allows us to refer the objects stored in GemFire data regions. This is very similar to SQL in syntax.

A basic query can be such as: SELECT DISTINCT \* FROM exampleRegion

GemFire’s QueryService provides methods to create the query object.

To manage the data serialization-deserialization, GemFire provides options other than Java serialization that gives a higher performance, provides greater flexibility for data storage and data transfer, also support for different languages.

With that in mind, GemFire has defined Portable Data eXchange(PDX) data format. PDX is a cross-language data format that provides a faster serialization and deserialization, by storing the data in the named field which can be accessed directly without the need of fully deserialization the object.

In GemFire, a function can reside on a server and can be invoked from a client application or another server without the need to send the code itself.

The caller can direct data-dependent function to operate on a particular data set or can lead an independent data function to work on a particular server, member or member group.

With continuous querying, the clients subscribe to server-side events by using SQL-type query filtering. The server sends all the events that modify the query results. The continuous querying event delivery uses the client/server subscription framework.

The syntax for a continuous query is similar to basic queries written in OQL. For example, a query which provides the latest stock data from Stock region can be written as:

SELECT \* from StockRegion s where s.stockStatus = ‘active’;

To get the status update from this query, an implementation of C QListener need to be attached with the StockRegion:

<cache>

<region name="StockRegion>

<region-attributes refid="REPLICATE"> ... <cache-listener>

<class-name>...</class-name> </cache-listener> ... </region-attributes>

</region>

</cache>

*SPRING DATA GEMFIRE SUPPORT*

To simplify configuration, spring data GemFire provides various annotations for configuring core GemFire components: