How to use MongoDB with Go

October 11, 2021 · 8 min read

The demand for applications that use NoSQL-based databases is on the rise, with many developers looking to learn how to integrate databases like MongoDB into applications built with their favorite language and frameworks.

In this tutorial, I'll teach you how to integrate MongoDB into Go applications seamlessly, by showing how to perform CRUD operations using the official Go driver for MongoDB, and providing code samples along the way.

Prerequisites

To follow and understand this tutorial, you will need the following:

- MongoDB installed on your machine
- Working knowledge of Go
- Go 1.x installed on your machine
- A Go development environment (e.g., text editor, IDE)

Getting started with MongoDB

The first step is to install mongo-go-driver, the official Go driver for MongoDB. It provides functionalities that allow a Go application to connect to a MongoDB database and execute queries.

Step 1: Set up your development environment

Create a new Go project in your text editor or IDE and initialize your `go.mod` file. You are free to use any name for your package:

```
go mod init mongo-with-golang
```

Step 2: Install the Go driver for MongoDB

Install the mongo-go-driver package in your project. In the terminal, type the following:

```
go get go.mongodb.org/mongo-driver/mongo
go get go.mongodb.org/mongo-driver/bson
```

Step 3: Create a MongoDB client instance

Import the Go driver package into your application, then create a MongoDB client instance for a database on port `27017` (MongoDB's default port).

Create a file named `main.go` and save the following code in it:

```
package main

import (
        "context"
        "go.mongodb.org/mongo-driver/mongo"
        "go.mongodb.org/mongo-driver/mongo/options"
        "go.mongodb.org/mongo-driver/mongo/readpref"
)

func main() {
        client, err := mongo.Connect(context.TODO(),
options.Client().ApplyURI("mongodb://localhost:27017"))
        if err != nil {
                panic(err)
        }
}
```

Here, you imported the `mongo` , `mongo/options` , and `mongo/readpref` modules from the Go driver into your application to connect to the local database.

Then, you created a client instance using the `mongo.Connect()` function and passed a Go `context` to it. Any time you make requests to a server (the database, in this case), you should create a context using `context.TODO()` that the server will accept.

Finally, you checked errors in the database connection using the `err` variable returned from calling `mongo.Connect()` . If the `err` value is not empty, it means there was an error (wrong credentials or connecting to a non-existent database), and you should terminate the application using `panic()` .

> *The `mongo.Connect` documentation contains more advanced configurations for creating a MongoDB client instance, including authentication.*

Step 4: Ping the MongoDB database

The MongoDB client provides a `Ping()` method to tell you if a MongoDB database has been found and connected.

Let's see how you can use it:

```
if err := client.Ping(context.TODO(), readpref.Primary()); err != nil {
        panic(err)
}
```

Here, you called the `Ping()` method and passed a `context` to it along with a primary read preference using `readpref.Primary()`, telling the MongoDB client how to read operations to the replica set members.

Then, you checked for errors using the `err` variable like we did earlier, and terminated the program using `panic()`, if required. If the code runs without any errors, it means the database connection is successful.

Step 5: Create a MongoDB collection instance

After you have connected to a MongoDB database, you need to create a `Collection` instance from the `client` instance that you will use to execute queries.

Add the following code to the `main.go` file to create a `Collection` instance retrieved from the `"users"` collection named `"testing"`:

- Use React's useEffect to optimize your application's performance
- Switch between multiple versions of Node
- Discover how to animate your React app with AnimXYZ
- Explore Tauri, a new framework for building binaries
- Compare NestJS vs. Express.js

```
usersCollection := client.Database("testing").Collection("users")
```

This code retrieves the `"users"` collection from the `"testing"` database in our local MongoDB database. If a database or collection does not exist before retrieving it, MongoDB will create it automatically.

Performing CRUD with MongoDB

Now that you have successfully established a connection to a MongoDB server and created a `Collection` instance, let's proceed to execute queries in our database from Go. This section covers how to insert, fetch, update, and delete data in a MongoDB database using the Go driver.

First, import the `bson` package we installed earlier into your project before working with data in MongoDB.

Add `"go.mongodb.org/mongo-driver/bson"` to your imports:

Creating new documents in MongoDB

To create new documents in a MongoDB collection, the database client provides an `InsertOne()` method that allows you to insert a single document, and an `InsertMany()` method to insert multiple documents.

Let's see how you can use them:

```
// insert a single document into a collection
// create a bson.D object
user := bson.D{{"fullName", "User 1"}, {"age", 30}}
// insert the bson object using InsertOne()
result, err := usersCollection.InsertOne(context.TODO(), user)
// check for errors in the insertion
if err != nil {
        panic(err)
}
// display the id of the newly inserted object
fmt.Println(result.InsertedID)

// insert multiple documents into a collection
// create a slice of bson.D objects
users := []interface{}{
        bson.D{{"fullName", "User 2"}, {"age", 25}},
        bson.D{{"fullName", "User 3"}, {"age", 20}},
        bson.D{{"fullName", "User 4"}, {"age", 28}},
}
```

Here, you created a `bson` object to store data you want to insert into the database, because the MongoDB driver requires you to prepare your data as `bson`. You can also create an array and slice of `bson` objects to store multiple values.

Then, you used the `InsertOne()` method to insert a single object and `InsertMany()` method to insert a list of objects into the database collection.

Finally, you checked if there was an error in the operation using the `err` variable returned by the method, and displayed the ID of the newly inserted documents using the `InsertedID` and `InsertedIDs` fields of the insertion results.

Reading documents from MongoDB

To retrieve documents from a MongoDB collection, the database client provides a `Find()` method that returns all documents that match a search filter, and a `FindOne()` method that returns only the first document that matches the filter.

Let's look at how you can use them:

```
// retrieve single and multiple documents with a specified filter using
FindOne() and Find()
// create a search filer
filter := bson.D{
        {"$and",
                bson.A{
                        bson.D{
                                {"age", bson.D{{"$gt", 25}}},
                        },
                },
        },
}

// retrieve all the documents that match the filter
cursor, err := usersCollection.Find(context.TODO(), filter)
// check for errors in the finding
if err != nil {
        panic(err)
}
```

Here, you created a search filter to query the database for documents with values greater than `25` in their `age` field. A filter defines the set of parameters MongoDB should use to match the documents in the database and retrieve them for the user.

Next, you used the `Find()` method to retrieve all the documents that match the search filter by providing a request context and search filter as arguments. The `Find()` method returns a `cursor` object representing the retrieved documents and an `error` variable containing any errors when querying the database.

After getting the resulting `cursor` object, you used the `cusor.All()` function to convert the cursor data to a slice of `bson` objects. Then, we checked for errors using the `err` variable and displayed the retrieved document in the terminal.

Then, you used the `FindOne()` method to retrieve the first document that matches the search filter. The `FindOne()` method returns an object you can convert to a `bson` object using the `Decode()` method.

Finally, you checked for errors in the `Find()` and `Decode()` operations using the `err` variable and displayed the retrieved document in the terminal.

You can also retrieve every document in a collection by matching the `Find()` method with an empty filter:

```
// retrieve all the documents in a collection
cursor, err := usersCollection.Find(context.TODO(), bson.D{})
// check for errors in the finding
if err != nil {
        panic(err)
}

// convert the cursor result to bson
var results []bson.M
// check for errors in the conversion
if err = cursor.All(context.TODO(), &results); err != nil {
        panic(err)
}

// display the documents retrieved
fmt.Println("displaying all results in a collection")
for _, result := range results {
        fmt.Println(result)
}
```

You should use `bson.D` objects when you care about the field order in the `bson` object (e.g., command and filtering documents), then use `bson.M` objects when you don't care about the order of the fields.

Updating documents in MongoDB

MongoDB provides two operations to change documents in a collection: `Update` and `Replace`. `Update` changes only specified fields in a document, while `Replace` overwrites existing data with new fields you provide.

The MongoDB driver also provides the following functions to change documents in a collection, they are:

- `UpdateByID()`
- `UpdateOne()`
- `UpdateMany()`
- `ReplaceOne()`
- `FindOneAndUpdate()`
- `FindOneAndReplace()`

Let's explore each of the functions starting with `UpdateByID()`, which updates the fields of a single document with a specified `ObjectID`:

```
// update a single document with a specified ObjectID using UpdateByID()
// insert a new document to the collection
user := bson.D{{"fullName", "User 5"}, {"age", 22}}
insertResult, err := usersCollection.InsertOne(context.TODO(), user)
if err != nil {
        panic(err)
}

// create the update query for the client
update := bson.D{
        {"$set",
                bson.D{
                        {"fullName", "User V"},
                },
        },
        {"$inc",
                bson.D{
                        {"age", 1},
                },
```

Here, you inserted a new document into the collection and created an update query that will set the `fullName` field of matched documents with `"User V"`, then increment the `age` field by `1`.

Next, you used the `UpdateByID()` function to update the specified document by providing a context, the `ObjectID` of the document you want to modify, and the `update` query to execute as arguments.

Finally, you checked for errors in the `update` operation using the `err` variable, and displayed the number of modified documents using the `UpdateResult` object returned from calling `UpdateByID()`.

Now, let's look at the `UpdateOne()` and `UpdateMany()` functions to update single and multiple documents that match a specified search filter:

```
// update single and multiple documents with a specified filter using
UpdateOne() and UpdateMany()
// create a search filer
filter := bson.D{
        {"$and",
                bson.A{
                        bson.D{
                                {"age", bson.D{{"$gt", 25}}},
                        },
                },
        },
}

// create the update query
update := bson.D{
        {"$set",
                bson.D{
                        {"age", 40},
                },
```

Here, you first created a search filter that matches documents with values greater than `25` in their `age` field. Then, you created an `update` query that changes the value of the `age` field to `40`.

Next, you used the `UpdateOne()` function to update the first document that matches the search filter by providing a context, the filter with which to match documents, and the `update` query to execute as arguments.

The `UpdateOne()` method returns an `UpdateResult` object containing information about the operation results, and an `error` variable containing any errors when updating the database.

Finally, you used the `UpdateMany()` function to update all the documents that match the search filter by providing the same arguments as the `UpdateOne()` function above.

Now, let's look at the `ReplaceOne()` function to overwrite the data in a document that matches a specified search filter:

```
filter := bson.D{{"fullName", "User 1"}}

// create the replacement data
replacement := bson.D{
        {"firstName", "John"},
        {"lastName", "Doe"},
        {"age", 30},
        {"emailAddress", "johndoe@email.com"},
}

// execute the ReplaceOne() function to replace the fields
result, err := usersCollection.ReplaceOne(context.TODO(), filter,
replacement)
// check for errors in the replacing
if err != nil {
        panic(err)
}
// display the number of documents updated
fmt.Println("Number of documents updated:", result.ModifiedCount)
```

Here, you created a search filter that matches documents with a value of `"User 1"` in their `fullName` field and a `bson` object containing the new data to store.

Then, you used the `ReplaceOne()` function to overwrite the data of the first document that matches the search filter by providing a context, the filter to match documents with, and the replacement data as arguments.

Finally, you checked for errors in the replace operation using the `err` variable and displayed the number of modified documents using the `UpdateResult` object returned from calling `ReplaceOne()`.

The `FindOneAndUpdate()` and `FindOneAndReplace()` functions perform the same operation as `FindOne()` and `ReplaceOne()`, but will return a copy of the document before modifying it.

Deleting documents from MongoDB

To delete documents from a MongoDB collection, the database client provides a `DeleteOne()` method to delete a single document and a `DeleteMany()` method to delete multiple documents in a collection.

Let's see how you can use them:

```
// delete single and multiple documents with a specified filter using
DeleteOne() and DeleteMany()
// create a search filer
filter := bson.D{
        {"$and",
                bson.A{
                        bson.D{
                                {"age", bson.D{{"$gt", 25}}},
                        },
                },
        },
}

// delete the first document that match the filter
result, err := usersCollection.DeleteOne(context.TODO(), filter)
// check for errors in the deleting
if err != nil {
        panic(err)
}
```

Conclusion

I hope this was a helpful guide to what can often be a challenging challenging task. The lack of straightforward resources on using MongoDB with Go requires developers to spend a lot of time exploring documentation. With this article as a reference guide, you can confidently integrate MongoDB into a Go application.

You can head over to the official MongoDB and Go driver documentation to explore more functionalities that MongoDB provides. Also, you can visit MongoDB University to build your skills and advance your career with courses and certifications.

Solomon Esenyi Follow

Python/Golang developer and Technical Writer with a passion for open-source, cryptography, and serverless technologies.

#go

Leave a Reply

Enter your comment here...