

Go (#golang) and MongoDB using mgo

October 14, 2012

After working in [node.js](#) last year, I've [switched to learning Go](#) instead, and I wanted to reprise my "[Node.js and MongoDB: A Simple Example](#)" post in [Go](#). Of all the [Go drivers](#) available for mongoDB, [mgo](#) is the most advanced and well-maintained. The example on the [mgo main page](#) is easy to understand:

1. Create a [struct](#) which matches the [BSON documents in the database collection](#) you want to access
2. Obtain a session using the [Dial function](#), which creates a connection object
3. Use the connection object to access a particular collection in your database:
 - o Searches load documents from the database into the struct
 - o Inserts and updates take data defined in a struct and create/update documents in the database

So for a collection named "Person", where a typical document looks like this:

```
{
  "_id" : ObjectId("502fbbd6fec1300be858767e"),
  "lastName" : "Seba",
  "firstName" : "Jun",
  "inserted" : ISODate("2012-08-18T15:59:18.646Z")
}
```

The corresponding Go struct would be:

```
type Person struct {
  Id          bson.ObjectId  "_id,omitempty"
  FirstName   string         "firstName"
  MiddleName  string         "middleName,omitempty"
  LastName    string         "lastName"
  Inserted    time.Time      "inserted"
}
```

It turns out the third field in each line, the string literal tag which is normally optional in a Go struct, is required here, because mgo won't find those fields in the database otherwise. It's also possible to convert database results directly into [json](#), which is useful for creating [API services](#) that output json. In that case, it's necessary to define both a bson tag and a json one, surrounded by backticks:

```
type Person struct {
  Id          bson.ObjectId  `bson:"_id,omitempty" json:"- "`
  FirstName   string         `bson:"firstName" json:"firstName"`
  MiddleName  string         `bson:"middleName,omitempty" json:"middleName,omitempty"`
  LastName    string         `bson:"lastName" json:"lastName"`
  Inserted    time.Time      `bson:"inserted" json:"- "`
}
```

The json tag follows the conventions of the built-in [Go json package](#): "-" means ignore, "omitempty" will exclude the field if its value is empty, etc. So far so good. But accessing different collections in a database means that for each one: it has its own struct defined, it has its own connection with the collection name specified, and an access function ([Find](#), [Insert](#), [Remove](#), etc.) which marshals/unmarshals those results. And the last step in particular can lead to a lot of code repetition. Inspired by [Alexander Luya's post](#) on [mgo-users](#), I've created a framework that allows for multiple access functions with a minimum of repetition. First, this function, which creates or clones the call to [Dial\(\)](#) as needed (this is very similar to what Alex posted):

```
var (
  mgoSession      *mgo.Session
  databaseName = "myDB"
)

func getSession () *mgo.Session {
  if mgoSession == nil {
    var err error
    mgoSession, err = mgo.Dial("localhost")
    if err != nil {
      panic(err) // no, not really
    }
  }
  return mgoSession.Clone()
}
```

Next, a higher-order function which takes a collection name and an access function prepared to act on that collection:

```

func withCollection(collection string, s func(*mgo.Collection) error) error {
    session := getSession()
    defer session.Close()
    c := session.DB(databaseName).C(collection)
    return s(c)
}

```

The `withCollection()` function takes the name of the collection, along with a function that expects the connection object to that collection, and can execute access functions on it. Here's how the "Person" collection can be searched, using the `withCollection()` function:

```

func SearchPerson(q interface{}, skip int, limit int) (searchResults []Person, searchErr string) {
    searchErr = ""
    searchResults = []Person{}
    query := func(c *mgo.Collection) error {
        fn := c.Find(q).Skip(skip).Limit(limit).All(&searchResults)
        if limit < 0 {
            fn = c.Find(q).Skip(skip).All(&searchResults)
        }
        return fn
    }
    search := func() error {
        return withCollection("person", query)
    }
    err := search()
    if err != nil {
        searchErr = "Database Error"
    }
    return
}

```

The `skip` and `limit` parameters are optional in that if `skip` is set to zero, it is effectively asking for all the results, and, similarly, if `limit` is set to an integer less than zero, it is ignored in the query that gets invoked inside the `withCollection()` function. So with that framework in place, making a variety of different queries on the "Person" collection reduces to writing simple (often one-line) BSON queries, as in the following examples. (1) Get all people whose last name begins with a particular string:

```

func GetPersonByLastName(lastName string, skip int, limit int) (searchResults []Person, searchErr string) {
    searchResults, searchErr = SearchPerson(bson.M{"lastName": bson.RegEx{"^"+lastName, "i"}}, skip, limit)
    return
}

```

(2) Get all people whose last name is exactly the given string:

```

func GetPersonByExactLastName(lastName string, skip int, limit int) (searchResults []Person, searchErr string) {
    searchResults, searchErr = SearchPerson(bson.M{"lastName": lastName}, skip, limit)
    return
}

```

(3) Find people whose first and last names begin with the particular strings:

```

func GetPersonByFullName(lastName string, firstName string, skip int, limit int) (searchResults []Person, searchErr string) {
    searchResults, searchErr = SearchPerson(bson.M{
        "lastName": bson.RegEx{"^"+lastName, "i"},
        "firstName": bson.RegEx{"^"+firstName, "i"}}, skip, limit)
    return
}

```

(4) Find people whose first and last names match with first and last names exactly:

```

func GetPersonByExactFullName(lastName string, firstName string, skip int, limit int) (searchResults []Person, searchErr string) {
    searchResults, searchErr = SearchPerson(bson.M{"lastName": lastName, "firstName": firstName}, skip, limit)
    return
}


```

et. cetera. As far as code repetition goes, however, this framework is not that efficient in that each collection requires its own `Search[Collection]()` function, where the only difference among the different functions is the type of the `searchResults` variable. It would be tempting to write something like this:

```
func Search (collectionName string, q interface{}, skip int, limit int) (searchResults []interface{}, searchErr string) {
    searchErr = ""
    query := func(c *mgo.Collection) error {
        fn := c.Find(q).Skip(skip).Limit(limit).All(&searchResults)
        if limit < 0 {
            fn = c.Find(q).Skip(skip).All(&searchResults)
        }
        return fn
    }
    search := func() error {
        return withCollection(collectionName, query)
    }
    err := search()
    if err != nil {
        searchErr = "Database Error"
    }
    return
}
```

Except this is where Go's strong typing gets in the way: *"there's no magic that would turn an interface{} into a Person"*, and so each `Search[Collection]()` function has to be written separately.

Archived from the original at <http://denis.papathanasiou.org/>

 Bitcoin Donate: [14TM4ADKJbaGEi8Qr8dh4KfPBQmjTshkZ2](https://blockchain.info/address/14TM4ADKJbaGEi8Qr8dh4KfPBQmjTshkZ2)