README.md Demo

Code to demo use of MonGolang in Jupyter notebooks. Highlights use of Aggregation, JSON Strings and Structures on the zips colllection.

Setup

In [1]:
```
import (
    "fmt"

    "go.mongodb.org/mongo-driver/bson"
        "go.mongodb.org/mongo-driver/mongo"
        "go.mongodb.org/mongo-driver/mongo/options"

    "github.com/ddgarrett/mongolang"
)
```

In [2]:
```
db := mongolang.DB{}
db.InitMonGolang("mongodb://localhost:27017").Use("quickstart")
```

Out[2]:
```
&{0xc0001e6a80 <nil> 0xc000a84300 quickstart}
```

Using JSON Strings

JSON strings can be passed to MongoDB calls anywhere a `bson.A` or `bson.D` struct can be passed.

This includes the filter and projection parms in `Find(...)` and `FindOne(...)` as well as others.

Note the use of Go's *raw strings*, strings enclosed in back ticks. Go raw strings allow embeding quotes and other special characters without having to escape them.

In [3]:
```
// Third largest zipcode in terms of population
db.Coll("zips").        // JSON string for filter state == "CA"
                        // and projection to eliminate the "loc" fiel
d
    Find(`{"state":"CA"}`, `{"loc":0}`).
    Sort(`{"pop":-1}`). // sort on descending "pop" using JSON string
    Skip(2).
    Limit(1).
    Pretty()
```

Out[3]:
```
{
    _id : 90650
    city : NORWALK
    pop : 94188
    state : CA
}
```

JSON Strings in Aggregate

Where JSON strings become even more helpful is in complex aggregation calls. The call below finds the three cities in the US with the highest population.

In [4]:
```
// Three largest cities
pipeline := `[
    { "$group":
        {
            "_id": { "State": "$state", "City": "$city" },
            "Pop": { "$sum": "$pop" }
        }
    },
    {"$sort": {"Pop":-1}},
    { "$limit": 3 }
]`

db.Coll("zips").Aggregate(pipeline).Pretty()
```

Out[4]:
```
{
    _id : [{State IL} {City CHICAGO}]
    Pop : 2452177
}
{
    _id : [{State NY} {City BROOKLYN}]
    Pop : 2300504
}
{
    _id : [{State CA} {City LOS ANGELES}]
    Pop : 2102295
}
```

Even More Complex Aggregate

Below is an even more complex pipeline. This pipeline finds, for each state, the cities with the highest and lowest populations. The results are then sorted by the population of the largest city and the first three are then printed.

In [5]:
```
pipeline = `
[
    { "$group":
       {
         "_id": { "state": "$state", "city": "$city" },
         "pop": { "$sum": "$pop" }
       }
    },
    { "$match": {"pop": {"$gt":10}}},
    { "$sort": { "pop": 1 } },
    { "$group":
       {
         "_id" : "$_id.state",
         "biggestCity":  { "$last": "$_id.city" },
         "biggestPop":   { "$last": "$pop" },
         "smallestCity": { "$first": "$_id.city" },
         "smallestPop":  { "$first": "$pop" }
       }
    },
   { "$project":
     { "_id": 0,
       "state": "$_id",
       "biggestCity":  { "name": "$biggestCity",
                         "pop": "$biggestPop" },
       "smallestCity": { "name": "$smallestCity",
                         "pop": "$smallestPop" }
     }
   },
   { "$sort": { "biggestCity.pop": -1 } },
   {"$limit": 3}
 ]
`

db.Coll("zips").Aggregate(pipeline).Pretty()
```

Out[5]:
```
{
    biggestCity : [{name CHICAGO} {pop 2452177}]
    smallestCity : [{name ANCONA} {pop 38}]
    state : IL
}
{
    biggestCity : [{name BROOKLYN} {pop 2300504}]
    smallestCity : [{name GRAND GORGE} {pop 13}]
    state : NY
}
{
    biggestCity : [{name LOS ANGELES} {pop 2102295}]
    smallestCity : [{name CHUALAR} {pop 12}]
    state : CA
}
```

Passing Slices for Results

So far the examples have shown examples which are slices (go version of resizable arrays) of bson.D. MonGolang also allows a slice to be passed to methods which normally return a slice array of bson.D documents. If a slice is passed the results will be placed in the passed slice and an empty bson.D slice will be returned.

This can be useful in two cases - returning the results as a bson.M (map) instead of the bson.D and when you want to save the results in a predefined go struct.

bson.M vs bson.D

bson.M, maps, do not provide an ordered set of {key,value} pairs. When you insert a new MongoDB document in a collection it is usually important to insert the fields in a consistent order. This makes the results of queries much more readable to humans. For example, in the previous results from our `Find(...)` , if the documents had been inserted using a map or read using a map, the order of the fields would vary.

When the documents are read into a bson.M slice, the same applies. Printing the individual map entries you'll see that the order of the fields vary.

In [6]:
```
// Third largest zipcode in terms of population

bsonM := []bson.M{}

_ := db.Coll("zips").
        Find(`{"state":"CA"}`).
        Sort(`{"pop":-1}`).
        Limit(3).
        ToArray(&bsonM)
```

In [7]:
```
for _,v := range bsonM {
    fmt.Println(v)
}
```

```
map[_id:90201 city:BELL GARDENS loc:[-118.17205 33.969177] pop:99568
state:CA]
map[city:LOS ANGELES loc:[-118.258189 34.007856] pop:96074 state:CA _
id:90011]
map[city:NORWALK loc:[-118.081767 33.90564] pop:94188 state:CA _id:90
650]
```

In fact, the order of the fields printed will vary even if we don't reread the data.

In [8]:
```
for _,v := range bsonM {
    fmt.Println(v)
}
```

```
map[_id:90201 city:BELL GARDENS loc:[-118.17205 33.969177] pop:99568
state:CA]
map[_id:90011 city:LOS ANGELES loc:[-118.258189 34.007856] pop:96074
state:CA]
map[_id:90650 city:NORWALK loc:[-118.081767 33.90564] pop:94188 stat
e:CA]
```

Contrast this with the bson.D that the previous examples used. In that case the order of the fields was always the same.

Why bson.M?

So if you can't depend on the order of the fields in bson.M, why use it? bson.M has one big advantage over bson.D - the field values can be easily accessed and set using the field name. This makes it much easier to print a formatted version of the results or use the results for another query. Addressing fields by key name is not easy with bson.D. With bson.M it's as simple as specifying the name of the field.

In [9]:
```
for _,v := range bsonM {
    fmt.Printf("%s, %s population %d\n", v["city"], v["state"], v["pop"] )
}
```

```
BELL GARDENS, CA population 99568
LOS ANGELES, CA population 96074
NORWALK, CA population 94188
```

BSON Annotated Struct

Another option is to use a slice of a struct defined in a Go program. In this example we only need to access the city, state and population so we create a Go struct to define those three fields. Note the use of annotations such as `bson:"state"` . These are needed if the name of the field in the results will be different from the name in the struct.

Important Note By convention fields in a go struct are not visible outside of the package if they are lowercase. Therefore all of the fields in our program defined struct **must be** uppercase or else the MongoDB Driver will not be able to access them!

In [10]:
```
type CityStatePop struct {
    State string `bson:"state"`
    City  string `bson:"city"`
    Pop int    `bson:"pop"`
}
```

Using the struct we can now access the fields in the results even more easily. When we use a struct like this the compiler will also ensure that we're using a valid name.

In [11]:
```
var csp []CityStatePop

db.Coll("zips").
    Find(`{"state":"CA"}`).
    Sort(`{"pop":-1}`).
    Limit(3).
    ToArray(&csp)


fmt.Println("Three most populous zip codes in California:")
for _, v := range csp {
    fmt.Printf("%s, %s : population %d \n", v.City, v.State, v.Pop )
}
```

```
Three most populous zip codes in California:
BELL GARDENS, CA : population 99568
LOS ANGELES, CA : population 96074
NORWALK, CA : population 94188
```

Finally - we close the database connection.

In [12]:
```
db.Disconnect()
```

MonGolang

MonGolang provides an easy to use program interface that adheres closely to the Mongo Shell command line interface. With Golang raw strings JSON can be easily used for simple to complex queries, allowing the syntax to mimic the Mongo Shell interface to the extent possible.

Although the lack of an ordered map in Golang does present some limitations, the use of additional capabilites in Golang to return results in a `bson.M` or custom struct helps overcome those.

Jupyter notebook provides a facility much like the Mongo Shell in terms of the ability to interactively explore a MongoDB database but with much more. Jupyter Notebook allows you to rerun a series of queries and include markdown text to further explain the results of the queries.