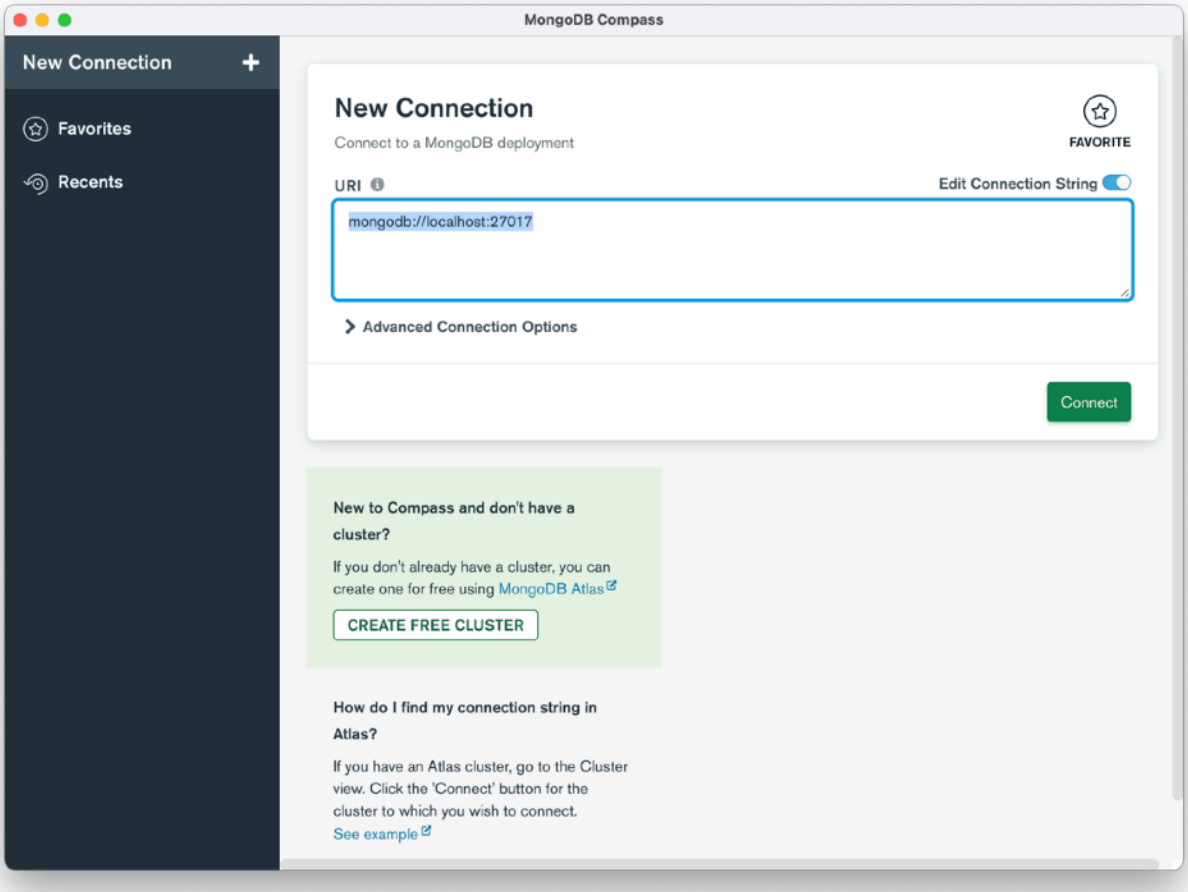
New MongoDB Example

A cool new thing comes with the MBS FileMaker Plugin 12.3 for us: You can connect in FileMaker to a Mongo DB server and create on this server databases, collections and documents with various functions, search in documents and delete them as needed. The special thing about Mongo DB is that it is not a relational database based on tables and relations, but its data has a JSON like structure. This allows you to make queries that were previously not possible due to the restriction of relationships or table boundaries. So find out who is longer in your company: the carpet in the warehouse or your trainee.

We add a new example file about Mongo DB to our plugin examples. I would like to show you this example here. To use this example you can download the free local community server of Mongo DB and install it on one of your computers. For my Mac, I can just use homebrew with a few Terminal commands. To work with the server we need a URI to locate it. If you also installed the Compass graphics environment when you installed the server, you can simply get it when you open Compass.



Normally you can reach your new local server under the following URI: mongodb://localhost:27017

Now you can use the example. The very first thing we want to do in FileMaker, of course, is to connect to the MongoDB server. To do this, you can enter the URI in the field that is provided and then click on the Connect button. This will start the Connection script. This script first checks if there is already a MongoDB object. If yes, then this object will be released. After that we create a new MongoDB object with the function MongoDB.New. With MongoDB.SetURI we set the URI to establish a connection with MongoDB.Connect. We can now work on this connection. For example, with MongoDB.DatabasesNames we can output a list that tells us which databases already exist on the server.

 # Connection in file MongoDB Blog

```
# Connect us to the Server
#
# Old connections are cleaned up
If [ $$MongoDB ≠ "" ]
	Set Variable [ $r ; Value:  MBS( "MongoDB.Release"; $$MongoDB ) ]
	// Set Variable [ $r ; Value:  MBS( "MongoDB.ReleaseAll" ) ]
End If
#
# Get a new reference number
Set Variable [ $$MongoDB ; Value: MBS( "MongoDB.New" ) ]
If [ MBS("IsError") ]
	Show Custom Dialog [ "Error" ; $$MongoDB ]
	Exit Script [ Text Result:    ]
End If
# Set the URI from the field
Set Variable [ $r ; Value: MBS( "MongoDB.SetURI"; $$MongoDB; MongoDB::URI ) ]
If [ MBS("IsError") ]
	Show Custom Dialog [ "Error" ; $r ]
	Exit Script [ Text Result:    ]
End If
# Connect to the server
Set Variable [ $r ; Value: MBS( "MongoDB.Connect"; $$MongoDB) ]
If [ MBS("IsError") ]
	Show Custom Dialog [ "Error" ; $r ]
Else
	Show Custom Dialog [ "Connection" ; "OK" ]
End If
```

Creating a new database is not an active process in Mongo DB. A database is created when it is used for the first time. For this reason it is enough to open a database with MongoDB.OpenDatabase. If the name has already been used once, the existing database is used, otherwise a new one is created as soon as the first data is stored in this database. For MongoDB databases there are three different terms you

have to distinguish: Database, Collection and Document. A database can have several collections. Collections contain the individual data series called documents. Collections help you to structure your database. The figure below will help you to distinguish between them.

Documents always belong to a collection, so before we can work with the data we have to specify which collection we want to work with. To do this we open the collection with MongoDB.OpenCollection. We can now enter data into this collection. For this we have the functions MongoDB.InsertOne. In our example we use it in the script Insert One to create an entry with a name that we have previously placed in the intended field.

```
Set Variable [ $r ; Value: MBS( "MongoDB.InsertOne"; $
$MongoDB;
JSONSetElement ( "{}" ; "Name" ; MongoDB::Name;
JSONString )") ]
```

Data is passed to the function using JSON. So we specify the key Name and enter the field value as the value.

Of course, we can not only write data, but also read it. The MongoDB.Find function helps us in this task. Besides the object reference number, which we have to specify in most MongoDB functions, we also specify a JSON here that tells us what we are looking for in the database, whether it is a specific name, a number that is less

than a specific value, or a combination of both. We have used some filtering in the example to show you how this can possibly look. If we want to get back all documents contained in the collection as result, we specify an empty JSON. The result is not displayed directly yet, but we can step through the result set with a cursor (MongoDB.CursorNext) and combine the single documents e.g. in a text output. If we want to know how many documents are present in the current collection, we call the function MongoDB.EstimatedDocumentCount. It returns the number of existing documents. If you want to change one or more records, you can use the functions MongoDB.UpdateOne and MongoDB.UpdateMany. Again, we have a JSON in which we can specify certain criteria by which the documents will be filtered. There is also a second JSON in which the update to be made in the document(s) is described. Set Variable [$r ; Value: MBS("MongoDB.UpdateOne"; $ $MongoDB; JSONSetElement ("{}" ; "Name" ; MongoDB::Name; JSONString); JSONSetElement ("{}"; "$set"; JSONSetElement ("{}" ; "Age" ; MongoDB::Age; JSONNumber); JSONObject))] As the names suggest we can use the function MongoDB.UpdateMany to update multiple records that match the selector JSON. With MongoDB.UpdateOne it is the first record in the collection that matches the criteria in the Selctor JSON. You can also replace whole documents with documents that may have completely different keys. You can see this in our example in the script Replace. Here a document that has a certain name is replaced by a document that does not have the key name. For example, where the old document still described a human being, clothes are now described in the same place.

```
Set Variable [ $r ; Value: MBS( "MongoDB.ReplaceOne"; $
$MongoDB;
JSONSetElement ( "{}" ; "Name" ; MongoDB::Name;
JSONString );
"{\"Typ\": \"clothes\", \"ArtNum\": 1234, \"InStock\": true,
\"Age\": 3}") ]
```

Both documents can coexist in the collection without any problems. This is what makes MongoDB so flexible. If we now have a document that describes a person with a name and an age and we have on the other side a clothing item that is described with an article number and an age, I can for example determine whether the clothing item is older than the person by formulating an appropriate query.

If you want to delete documents, we have two functions that delete one or more documents at the same time. Again, we determine which records should be deleted using a JSON that gives us the criteria that the documents should match.

This and more awaits you in our new example. If you are interested, download the latest plugin version next week (12.4pr1) and try it out.

I wish you a lot of fun with it.
by Stefanie Juchmes