WHAT IS A DATABASE AND MONGODB

Bu kısımda database nedir ondan bahsedeceğiz, ve web application'ımız için neden database gerekiyor bunu açıklayacağız.

Database basitçe structured set of data held in the computer olarak tanımlanabilir. Zaten bildiğin bir şey, temelde database uygulama için gerekli verilerin daha sonra kullanılmak üzere saklandığı yer/yapı.

Dört işlem yapan API'ımızın herhangi bir veriyi saklamaya, veya authenticiation'a vesaire ihtiyacı yoktu, ancak buna ihtiyaç olan uygulamalar gerekebilir. API'mızı database ile haberleştirerek, dinamik uygulamalar yaratabiliriz, yani kullanıcıların yeni veri girişi yapabildikleri, bu verilerin saklandığı, daha sonra istenilirse kullanılabildiği uygulamalar.

Bu arada çoğu uygulama için database gerekecek, yani bizim yaptığımız dört işlem yapan API'ı bile piyasaya sunmaya karar veresek database'e ihtiyacımız olacaktır, hangi kullanıcının giriş yaptığını, hangi kullanıcının ne zaman hangi request'i kaç kez gönderdiğini store etmek isteriz ki kullanıcıyı ona göre fiyatlandırabilelim.

Bunun yanında database'i API'mızın sürekli yapması gereken işlemlerden kurtarmak için de kullanabiliriz, işlemin sonucunu database'e kaydederiz ve herseferinde hesaplama yapmak yerine database'den çekmek daha pratik olabilir.

İki temel tip database'den söz edilebilir: SQL ve No-SQL biz burada MongoDB kullanacağız bu da bir çeşit No-SQL database olacak.

Daha önce yaptığımız dört işlem API'nı geliştireceğiz, kullanıcı adı ve password kontrolü de olacak ve her kullanıcıya belirli sayıda request hakkı verilecek vesaire.

MONGODB INTRODUCTION

Şimdi MongoDB'den bahsedeceğiz.

MongoDB is a cross-platform, document oriented database that provides, high performance, high availability and easy scalability. MongoDB works on concept of collection and document.

Angular ile firebase uygulamaları yaparken zaten bir No-SQL database kullanmıştın, temel mantığı biliyorsun.

Database is a physical container for collections. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple databases.

Yani bir MongoDB Server'i içerisinde genelde birden fazla database olur, her bir database içerisinde de bir veya daha fazla collections olur, her her bir collection içerisinde de documents olur.

Collection is a group of MongoDB documents. It is the equivalent of an RDBMS table. A collection exists withing a single database. Collections do not enforce a schema. Documents within a collection can have different fields. Typically, all documents in a collection are of similar or related purpose.

A **document** is a set of key-value pairs. Document have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

Document'i bir json file olarak düşünebilirsin, key-value pairs'den oluşan bir yapı. Örneğin bir document, ID, Username ve Password keylerine karşılık values tutabilir, her bir user için bu yapıda ayrı bir document bir users collection'ı içerisinde tutulabilir. Sonuçta users bilgileri bu collection içinde tutulmuş olur.

Ayrıca bir document'in bir value'su olarak bir başka document tutulabilir buna Embedded sub-document deriz, firebase'de document value olarak yeni bir collection da oluşturulabiliyordu.

Relation to Normal DB

RDBMS	MongoDB
Database	Database
Table	Collection
Row	Document
Column	Field
Table Join	Embedded Documents
Primary Key	Default key _id is given by MongoDB

Yukarıdaki tabloda MongoDB ile normal (sanırım SQL) database arasındaki bazı ilişkiler yer alıyor.

SQL database altında bir veya daha fazla Table yer alır. Bu Table's MongoDB'da collections'a denk gelir.Table'ın her bir row'u ayrı bir girdi tutar, bu da MongoDB için ayrı bir document'a denk gelir. Her bir girdinin birden fazla column'u yani field'i vardır.

Relational databases'de table join kavramı mongodb için embedded documents'e denk gelir, buna bakacağız. Tables için her bir girdi'nin unique bir key'i olur, bu işlem mongodb için her document için verilen bir key_id ile çözülür.

Aşağıda bir document örneği var, value olarak bir başka document kullanılabildiğine dikkat et.

_id Field

Yukarıda da görüğün gibi her document'in bir _id key'i ve value'su var bu unique bir value olacak.

_id 12 byte (?) hexadecimal number which assures the uniqueness of every document. Bu değeri document yaratılırken kendimiz set edebiliriz, biz etmezsek de mongodb kendisi set eder.

Mongodb kendisi set ederse, dediğimiz gibi 12byte set eder, 12 byte'ın 4 byte'ı current timestamp, 3 byte'ı machine id, 2 byte'ı process id of mongodb server, last 3 bytes are simple incremental values.

MongoDB vs RDBMS

MongoDB databases'da schema design olmaz, rdbms'de olduğu gibi tables arası relations söz konusu değildir. Girdiler (documents) farklı fileds'e sahip olabilir.

MongoDB için complex joins söz konusu değildir, SQL kadar powerful olmasa da MongoDB databases üzerinde de document base query langauge kullanarak dynamic query yapılabilir.

MongoDB ile scaling is very easy, yani uygulama popüler olursa database'i büyütmek gayet kolay.

INSTALLING MONGODB ON THE UBUNTU VM

https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/

linkini kullanarak virtual machine'imiz üzerine mongodb yüklüyoruz. Instructions'ı takip ediyoruz. Gerekli paketleri yüklüyoruz.

Test etmek içinse:

You can start the mongod process by issuing the following command:

```
sudo systemctl start mongod
```

Verify that MongoDB has startes successfully:

```
sudo systemctl status mongod
```

As needed, you can stop the mongod process by issuing the following command:

```
sudo systemctl stop mongod
```

You can restart the mongod process by issuing the following command:

```
sudo systemctl restart mongod
```

Start a mongo shell on the same host machine as the mongod. You can run the mongo shell without any command-line options to connect to a mongod that is running on your localhost with default port 27017:

```
mongo
```

Bu shell'l çalıştırdıktan sonra db.help() ile farklı db komutlarını görebilirim, db.stats() ile current mongosb session'ın db bilgilerini görebilirim.

Anladığım kadarıyla MongoDB için gerekli packages'i install ettikten sonra local olarak mongodb databases oluşturabiliyoruz. Yani firebase'de olduğu gibi remote bir database kullanmak yerine, bir başka container'da çalışacak bir mongodb database yaratacağız. Daha sonra hem web service'ini hem de db service'ini aynı server'da deploy edeceğiz.

Web service'imiz remote bir db ile haberleşmeyecek, aynı server üzerindeki bir başka container'da çalışan bir db ile haberleşecek. Bunların detayına bakacağız.

CREATING AND DELETING DATABASES

Şimdi mongoDB session'ı üzerinde nasıl bir database create/delete ederiz ona bakacağız.

MongoDB database'inin çok kesin bir schema'sı yoktur, çünkü bildiğimiz gibi aynı collection'ın documents'ı birbirinden farklı olabilir, yani esnek bir schema'dan bahsedilebilir, bu schema'yı design ederken aşağıdaki unsurlara dikkat etmeliyiz.

Some considerations while designing Schema in MongoDB

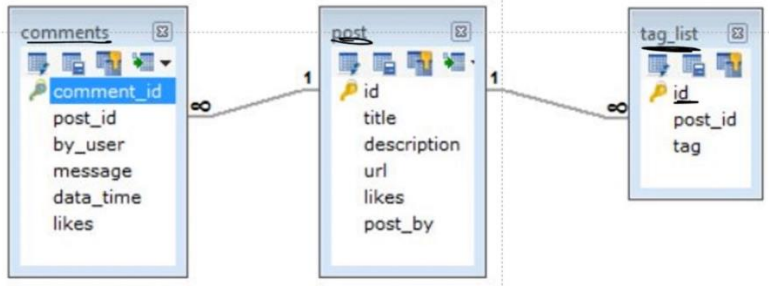
- Design your schema according to user requirements.
- Combine objects into one document if you will use them together. Otherwise separate them (but make sure there should not be need of joins).
- Duplicate the data (but limited) because disk space is cheap as compare to compute time.
- Do joins while write, not on read.
- Optimize your schema for most frequent use cases.
- Do complex aggregation in the schema.

fatkun

Example

- Suppose a client needs a database design for his blog/website and see the differences between RDBMS and MongoDB schema design. Website has the following requirements:
- Every post has the unique title, description and url.
- Every post can have one or more tags.
- Every post has the name of its publisher and total number of likes.
- Every post has comments given by users along with their name, message, data-time and likes.
- On each post, there can be zero or more comments.

- In RDBMS schema, design for above requirements will have minimum three tables:



RDBMS database yukarıdaki gibi olur, üç farklı table olacak comments, posts ve tag_lists için.

MongoDB için ise durum daha farklı olacak, tek bir posts collection'ı olur, bu her bir document bir post'un tüm bilgilerini içerir:

Creating a Database

Bir database create edebilmek için öncelikle önceki başlıkta gördüğümüz gibi bir mongoDB session'ının başlamış olması gerekli. Session başladıktan sonra da **mongo** command'i ile bir shell başlatıyoruz, daha sonra aşağıdaki command'leri kullanabiliriz.

Bundan sonra

```
use DB_NAME
```

ile yeni bir db yaratabiliriz. Eğer halihazırda bu isimde bir db varsa switch to this db, eğer böyle bir database daha önce oluşturulmamışsa, create and switch to that db.

To check the currently selected database use the command:

```
db
```

To see the list of all databases, use command. Yalnız unutma bir database'in bu listede görünebilmesi için en az bir document eklenmesi gerek.

```
show dbs
```

Database'e basitçe document eklemek için:

```
db.collection_name.insert({"key":"value"})
```

Eğer session üzerinde shell açıkken, herhangi bir database yaratmadıysak default olarak test database'i aktif olur. Collections bu database' eklenecektir.

Dropping a Database

Biraz evvel mongoDB üzerinde yeni bir db oluşturmayı gördük, şimdi bir database'i nasıl sileriz onu göreceğiz. Collection veya documents işlemlerini daha sonra detaylı göreceğiz. Şuan en genel katmadan bir db oluşturup silmekten bahsediyoruz:

```
db.dropDatabase()
```

Yukarıdaki komut, selected database'i siler, eğer herhangi bir db selected değilse, test database'i silinir.

Yani örneğin yeni oluşturduğumuz newdb database'ini silmek istersek, önce switch ederiz sonra drop:

```
use newDB
db.dropDatabase()
```

CREATING AND DELETING COLLECTIONS

MongoDB session'ı açık, shell açık, databases yaratmayı ve silmeyi gördük, aşağıdaki commands kullanılabiliyor.

To create a collection on the selected database:

```
db.createCollection(name,options)
```

Options is a document and is used to specify configuration of collection. Bu options içerisinde max size of the collection vesaire gibi şeyleri set edebiliriz. Options parametresi opsiyoneldir, ancak name parametresi kesinlikle verilmeli.

Şimdi collections yaratırken kullanılabilecek bazı options'a göz atalım:

Field	Type	Description
Capped	Boolean	(Optional) If true, enables a capped collection. Capped collection is a fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size. If you specify true, you need to specify size parameter also.
autoIndexId	Boolean	(Optional) If true, automatically create index on _id field.s Default value is false.
Size	Number	(Optional) Specifies a maximum size in bytes for a capped collection. **If capped is true, then you need to specify this field also.**
Max	Number	(Optional) Specifies the maximum number of documents allowed in the capped collection.

Capped:

Eğer true verilirse, capped collection yaratmış oluruz, yani fixed size bir collection yaratırız, collection'ın max size'ı dolduktan sonra yeni entries en eski entries'den başlanarak overwrite edilir.

Size:

Byte cinsinden max size of a capped collection'ı belirtir. Capped true ise bu field'in specify edilmesi zorunludur.

autoIndexId:

True ise _id field'i için otomatik olarak index yaratır, default olarak false'tur.

Max:

Capped collection için max document size'ı belirtir.

Collection Creation Example

Test db üzerinde bir collection yaratalım:

```
use test
db.createCollection("myCollection")
```

Yaratılan collections'ı görmek için:

```
show collections
```

Peki ya options ile birlikte bir collection create etmek istersek:

```
db.createCollection("mycol", {capped : true, autoIndexId : true, size :
6142800, max : 10000})
```

Ancak hatırlarsan bir önceki kısımda bir db'ye document insert etmeye çalışırken zaten collection da yaratmıştık, yani collection aşağıdaki gibi de yaratılabiliyor:

```
db.colname.insert({"key" : "value"})
```

Eğer colname ismindeki collection daha önce yaratılmamışsa, otomatik olarak yaratılacak ve ilgili document bu collection'a eklenecek.

Dropping a Collection

Aşağıdaki command ile ilgili collection ve içerisindeki tüm documents silinecektir.

```
db.collection_name.drop()
```

MongoDB Data Types

- MongoDB supports many datatypes. Some of them are:
- **String** – This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.
- **Integer** – This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
- **Boolean** – This type is used to store a boolean (true/ false) value.
- **Double** – This type is used to store floating point values.
- **Min/ Max keys** – This type is used to compare a value against the lowest and highest BSON elements.

- **Arrays** – This type is used to store arrays or list or multiple values into one key.
- **Timestamp** – ctimestamp. This can be handy for recording when a document has been modified or added.
- **Object** – This datatype is used for embedded documents.
- **Null** – This type is used to store a Null value.
- **Symbol** – This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.

- **Date** – This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.
- **Object ID** – This datatype is used to store the document's ID.
- **Binary data** – This datatype is used to store binary data.
- **Code** – This datatype is used to store JavaScript code into the document.
- **Regular expression** – This datatype is used to store regular expression.

Inserting Documents to Collections

Bir collection'a bir document insert etmek için daha önce gördüğümüz **insert()** methodunu veya **save()** methodunu kullanabiliriz:

```
db.collection_name.insert(document)
```

Örnek:

```
db.mycol.insert({
   _id: ObjectId(7df78ad8902c),
   title: 'MongoDB Overview',
   description: 'MongoDB is no sql database',
   tags: ['mongodb', 'database', 'NoSQL'],
   likes: 100
})
```

Inserting Multiple Documents in a Single Query

Tek seferde birden fazla document insert etmek istersek, **insert()** methodu içerisine array of documents verebiliriz.

QUERY DOCUMENTS IN MONGODB

Collections'a insert edilen documents'ı nasıl query edebiliriz bu kısımda buna bakacağız.

To query data from MongoDB collection, we need to use MongoDB's **find()** method.

```
db.collection_name.find()
```

Bu metod ilgili collection'ın tüm documents'ini non-structured bir biçimde display eder. Eğer formatted bir display istersek, pretty() methodundan yararlanabiliriz:

```
db.collection_name.find().pretty()
```

Usage of find()

Find() methodu içerisine, bir document verilir ve bu document'a göre bir arama yapılır. Eğer find() içerisine boş bir document verirsek**: find({})** ilgili collections içerisindeki tüm documents return edilir.

Diyelim ki, ilgili documents içinde name field'i "ahmet" olan tüm documents'i return etmek istiyoruz o halde, find içerisine aşağıdaki gibi bir search document'i veririz:

```
db.collection_name.find({"name":"ahmet"}).pretty()
```

findOne()

find ilgili condition'ı sağlayan tüm documents'i return ederken, findOne() yalnızca bir tanesini return eder.

Querying Documents

Yukarıda da bahsedildiği gibi document query etmek için **find()** methodu kullanılıyor, bu find methodu içerisine bir **filter** alıyor, bu filter belli başlı conditions'ı temsil ediyor ve find methodu bu conditions'ı sağlayan documents'i return ediyor.

Aşağıda bazı temel query örnekleri var:

<u>Equality</u>

Yukarıda da gördüğümüz gibi specific bir field'i specific bir value'ya eşit olan documents'i getir diyebiliriz:

```
db.collection_name.find({"name":"ahmet"}).pretty()
```

Less Than

Numeric fields için, bu kullanılabilir örneğin likes field'i 50'den küçük olan documents return edilsin diyebiliriz:

```
db.collection_name.find({"likes":{$lt:50}}).pretty()
```

Less Than Equals

Yukarıdakine çok benziyor < ilişkisi yerine <= ilişkisi kurmuş oluyoruz:

```
db.collection_name.find({"likes":{$lte:50}}).pretty()
```

Greater Than

```
db.collection_name.find({"likes":{$gt:50}}).pretty()
```

Greater Than Equals

```
db.collection_name.find({"likes":{$gte:50}}).pretty()
```

Not Equals

```
db.collection_name.find({"likes":{$ne:50}}).pretty()
```

AND Operator

Yukarıda find methodu içerisine geçirebileceğimiz belli başlı bazı filters'ı gördük, bu filters temelde bir documents olarak find methoduna veriliyordu.

AND operatörü kullanarak birden fazla filter'ı aşağıdaki gibi birleştirebiliriz.

```
db.collection_name.find(
        $and:[
                {"name":"ahmet"},{"likes":{$gt:50}}
        ]
).pretty()
```

Yukarıdaki query ile, name alanı ahmet olan ve likes alanı 50'den büyük olan tüm documents return edilecek.

OR Operator

And operatörü ile çok benzer şekilde and yerine or yazarsak or operator'ı kullanılabilir.

```
db.collection_name.find(
        $or:[
                {"name":"ahmet"},{"likes":{$gt:50}}
        ]
).pretty()
```

UPDATING, SORTING AND LIMITING DOCUMENTS

Collection içerisindeki bir document'i update etmek için **update()** ve **save()** methodları kullanılabilir.

UPDATE DOCUMENTS

Update methodu, existing document'in field value'larını update ederken, save methodu var olan document'i yeni bir document ile replace etmek için kullanılır.

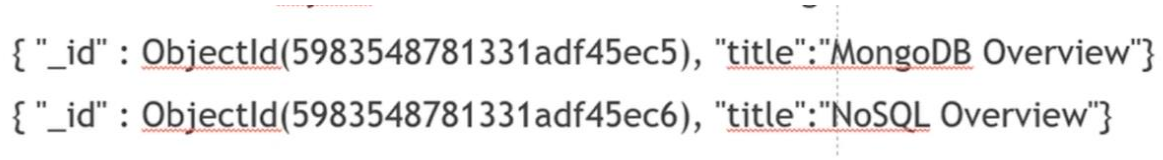
```
db.collection_name.update(selection_criteria, updated_data)
```

Selection_criteria bir filter olacak, hangi record/records'un update edileceğini belirtecek, updated_data ise hangi alanların nasıl update edileceğini belirtecek.

Birden fazla document'i update etmek istersek, örneklerde de göreceğimiz gibi multi parametresini true olarak geçirmeliyiz yoksa filter birden fazla doc seçse de tek bir document update edilecektir.

UPDATE DOCUMENTS EXAMPLE

Diyelim ki **mycol** isminde aşağıdaki gibi bir collection'ımız var:



Şimdi ilk document'in title alanını update edelim:

```
db.mycol.update( {'title':'MongoDB Overview'}, {$set: {'title':'New MongoDB Tutorial'}})
```

Title'ı şu şu olan document'ın title'ını şu şu olarak set et demiş olduk.

Updating Multiple Documents:

Selection criteria filter'ı birden fazla document seçse bile default olarak update sadece tek bir document'i update eder, multiple document update için update methoduna aşağıdaki gibi bir parametre geçirmeliyiz.

```
db.mycol.update( {'title':'MongoDB Overview'}, {$set: {'title':'New MongoDB Tutorial'}}, {multi:true})
```

DELETE DOCUMENTS

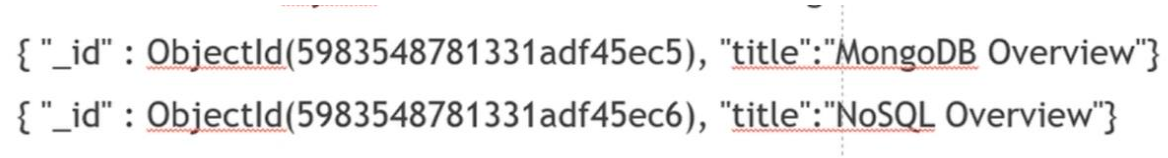
Document deletion işlemi ise **remove()** methodu ile yapılır.

Remove methodu iki parametre alır, ilki deletion criteria, yani aynen find veya update'de olduğu gibi bir filter. İkinci parametre ise, multiple document removal'ı mümkün kılan bir flag bu true olarak set edilirse, criteria'yı sağlayan tüm documents silinecektir.

```
db.colname.remove(deletion_criteria, justone)
```

DELETE DOCUMENTS EXAMPLE

Diyelim ki **mycol** isminde aşağıdaki gibi bir collection'ımız var:



Şimdi title'ı MongoDB Overview olan documents'i remove edelim:

```
db.colname.remove({'title':'MongoDB Overview'})
```

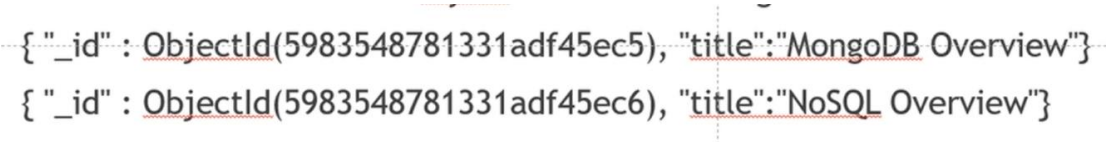
PROJECTION

Projection bir document'in sadece gerekli field'lerini seçme olayına denir. Yani örneğin bir document'in 5 field'i varsa ancak ben sadece 3 field'i görmek istiyorsam, projection ile sadece üç field'i seçmem gerek.

Bunu yapmak için find() methodunun içine filter'ın yanında bir başka parametre daha veririz, bu ikinci parametre filter'ı sağlayan documents'in hangi field'lerini retrieve etmek istediğimiz belirtir.

PROJECTION EXAMPLE

Aşağıdaki gibi bir mycol collection'ım olsun.



Şimdi diyelim ki, tüm documents'i return etmek istiyoruz ancak, sadece title field'i ile return etmek istiyoruz. O halde find() içerisine ilk paramere {} filterı verilecek bu tüm record'u return eder, ikinc, parametre içerisinde ise title:1, _id:0 verilecek böylece hangi alanların dahil edileceğini söylemiş oluruz:

```
db.mycol.find({},{'title':1, '_id':0})
```

LIMIT

Return edilen records'a bir limit koymak için **limit()** methodu kullanılır. Yani find methodundan sonra limit methodu kullanılırsa sonuçta return edilecek record/document sayısı kısıtlanmış olur.

```
db.colname.find().limit(NUMBER)
```

SORT

Return edilen record'u sort etmek için **sort()** methodunu kullanırız. Bu method içerisine parametre olarak bir document alır, bu document içerisinde, sort edilmek istenen field'e karşılık sort order no yer alır.

1:Ascending, -1:Decending order.

Örneğin aşağıdaki yapı, find sonucunda return edilen record'u likes alanına göre ascending olarak sıralar.

```
db.colname.find().sort({"likes":1})
```

BUILDING A SIMPLE APPLICATION

Şimdiye kadarki kısımda MongoDB'nin işleyişinden, temel komutlardan bahsettik. Aşağıdaki konulardan bahsettik.

- DB: create, delete
- Collection: create, delete
- Documents : Insert, Delete, Update
- Documents: Query

Ancak tüm bunları, terminal üzerinden bir MongoDB session başlattıktan sonra, bir shell açarak denedik.

Şimdi ise, flask üzerinden MongoDB ile nasıl interact edebileceğimize bakacağız!

Hatırlarsan, daha önceki kısımlarda docker kullanarak bir web app build etmiştik, yani dört işlem yapan api'ımızı dockerize etmiştik, ubuntu üzerinde docker engine yardımıyla kendine has bir container içerisinde bu "web" service'ini çalıştırabiliyorduk.

Şimdi ise, aynı project içerisinde yeni bir container içerisinde bir database uygulaması build edelim.

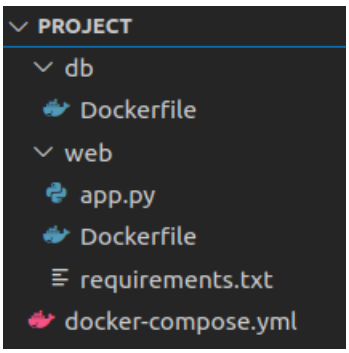
Amacımız, daha önce build ettiğimiz ve bir container içerisinde çalışan "web" flask uygulaması ile, şimdi oluşturacağımız yeni bir container içerisinde çalışacak olan "db" uygulamasını(aslında bir database) haberleştirmek olacak.

Mongo db'yi flask app içerisinde kullanabilmek için, pymongo package'i kullanılacak. Bu package'in aynen flask, flask_restful gibi import edilmesi gerekecek, aynı yerden import edilecek çünkü bu paketi kullanacak olan container "web" container'ı olacak.

Bu paketi kullanırken bazı mongoDB işlemleri shell işlemlerinden biraz daha farklı olacak, ama genel hatlarıyla benzer olacak ve mantık aynı olacak.

Şimdi uygulamaya geçelim bunun için daha önce kullandığımız proje klasörüne konumlanacağız.

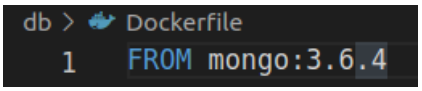
Bu klasörde "web" container'ını hazırlamıştık, docker compose farklı container'ları kontrol etmek ve aralarındaki haberleşmeyi sağlamak için kullanılıyordu.



1.

Gördüğümüz gibi ilk adım, db klasörüne konumlandıktan sonra touch Dockerfile diyerek burada bir dockerfile oluşturmak, bu dockerfile içerisine ilgili “db” container’ını set etmek için gerekli instructions’ı yazıyorduk:

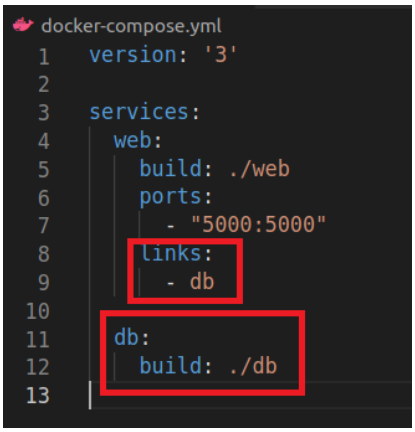
Burada sadece aşağıdaki satırı yazdık, bu satırla yapılan şey, docker hub üzerinde ilgili image’ı import etmek olacak, bu image ilgili container içerisinde bir mongoDB database çalıştırmak için gerekli herşeyi yapacak.



2.

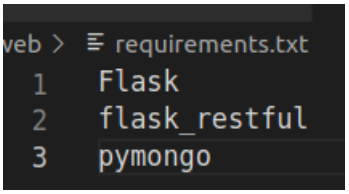
Şimdi sıradaki adım, bu “db” container’ını bir service olarak docker-compose.yaml dosyası içerisinde belirtmek, böylelikte console üzerinden hem “web” hem de “db” service’lerini build ve run edebileceğim.

Aşağıda görüldüğü gibi db service’ini build et diyorum ve directory veriyorum, bunun yanında “web” service’i “db” service’ine bağlı olarak çalışacağı için burada links belirttik, böylece “db” container’ı çalışır duruma gelmeden, “web” service’i çalışmayacaktır.



3.

Flask uygulamaıs içinden bu “db” ismindeki mongoDB server’ına ulaşmamız gerekecek bunun için de “web” container’ında flask, flask_restful’un yanında **pymongo** package’i kurulu olmalı, bunun için “web” container’ının **requirements.txt** dosyasını aşağıdaki şekilde güncelliyorum:



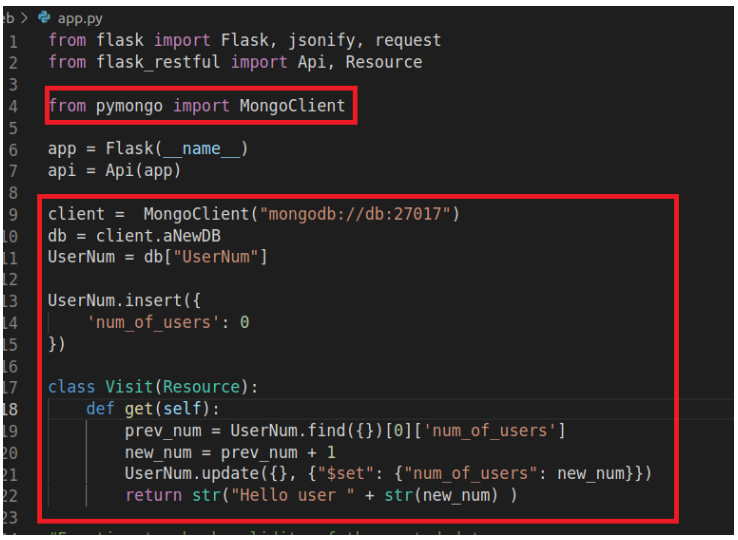
4.

Artık flask app'imiz yani app.py dosyası içerisinden, pymongo paketi aracılığı ile bu "db" mongoDB server'i ile iletişime geçebiliriz.

Bunu yapmak için ilk olarak aşağıdaki gibi, pymongo paketi üzerinde MongoClient objesi import ediliyor.

Daha sonra MongoClient objesi içerisine, aşağıdaki gibi "db" yani yaratılan mongoDB service'inin ismi ve default port no veriliyor. Sonuçta elimizde "db" service'i ile bağlantı kurabilen bir client objesi olmuş oluyor.

Bu client objesi üzerinde "db" service'i içerisinde mongoDB databases oluşturabilir, collections ve documents işlemleri yapabiliriz.



Yukarıda örnek olması için çok basit bir işlem yapılmış. Bunu açıklayalım, öncelikle client üzerinden **aNewDB** isminde yeni bir database create edilmiş, shell üzerinden bu işlemin biraz daha farklı yapıldığını hatırla ama mantık aynı.

Daha sonra oluşturululan bu aNewDB database'i üzerinden bir **UserNum** collection'ı oluşturulmuş. Hemen ardından da bu collection'a tek bir document eklenmiş, bu document içerisinde tek bir field tutuyor.

Ardından daha önce olduğu gibi add, subtract vb. gibi yeni bir Resource tanımlanmış, temelde bu resource'a get methodu ile ulaşılabilecek ve gelen get request için bu resource'a kaç kere ulaşıldığını return edecek, bunu yapmak için de "db" service'inden yaralanacak.

Görüdlüğü üzere yapılan, her request geldiğinde, ilgili collection'ın document'ine ulaşarak userCount'ı almak ve daha sonra bunu bir arttırarak, bu document'i update etmek, sonucunda da yeni count'ı response olarak yollamak.

Elbette hatırladığımız üzere, bu API resource'unun bir endpoint'i olmalıydı, bunu da aşağıdaki gibi set ediyoruz.

```
api.add_resource(Add,"/add")
api.add_resource(Subtract,"/subtract")
api.add_resource(Multiply,"/multiply")
api.add_resource(Divide,"/divide")
api.add_resource(Visit,"/visit")
```

Artık uygulamamız tamamıyla hazır, bu projeyi build ve run ettiğimizde docker üzerinde iki farklı container çalışacak biri "web" diğeri "db" service'imiz olacak ve "web" service'i "db" service'ini kullanarak request'leri handle edecek.

Bu build ve run işlemini yapmak için docker-compose.yml dosyası ile aynı dizindeyken aşağıdaki commands'i run ediyorduk:

```
sudo docker-compose build
```

```
sudo docker-compose up
```

Artık localhost:5000/visit'e aşağıdkai gibi request yollarsak, aşağıdaki response'u alırız, her request için number bir artacaktır!