WEB DEVELOPMENT USING MEAN STACK

Unit 5 - Connecting Angular js with MongoDB

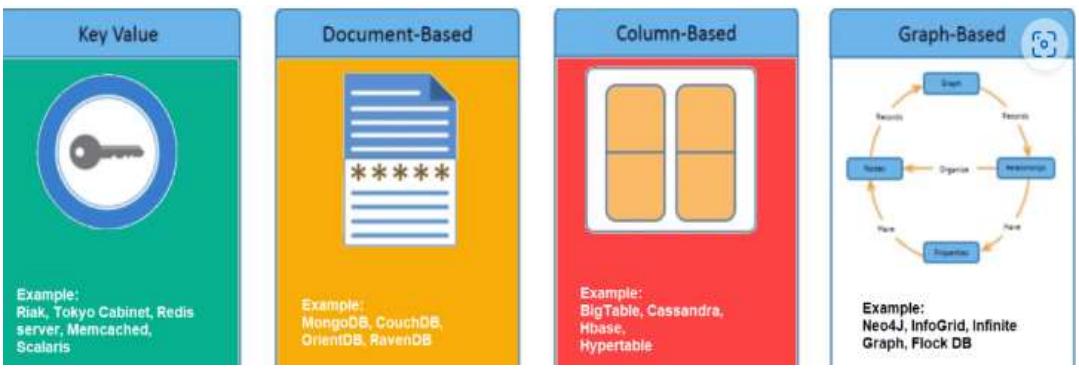
NoSQL is a non-relational DBMS,that does not require a fixed schema, avoids joins, and is easy to scale. NoSQL database is used for distributed data stores with humongous data storage needs. NoSQL is used for Big data and real-time web apps. For example companies like Twitter, Facebook, Google that collect terabytes of user data every single day.

NoSQL database stands for “Not Only SQL” or “Not SQL.”

Carl Strozz introduced the NoSQL concept in 1998.

Traditional RDBMS uses SQL syntax to store and retrieve data for further insights. Instead, a NoSQL can store structured, semi-structured, unstructured and polymorphic data.

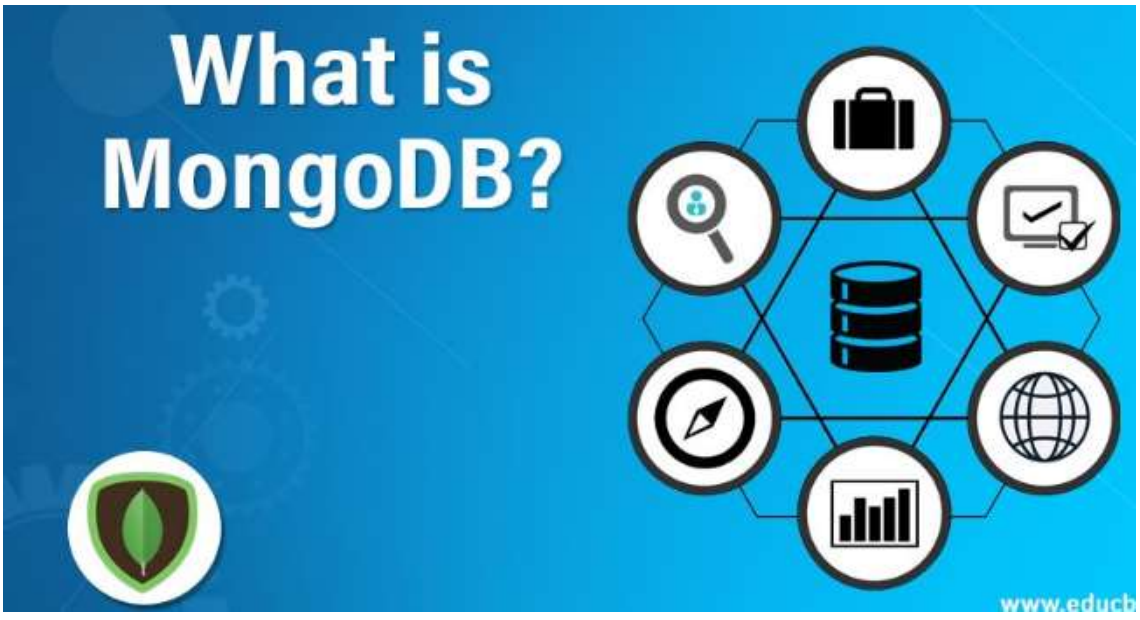
There are mainly four categories of NoSQL databases.



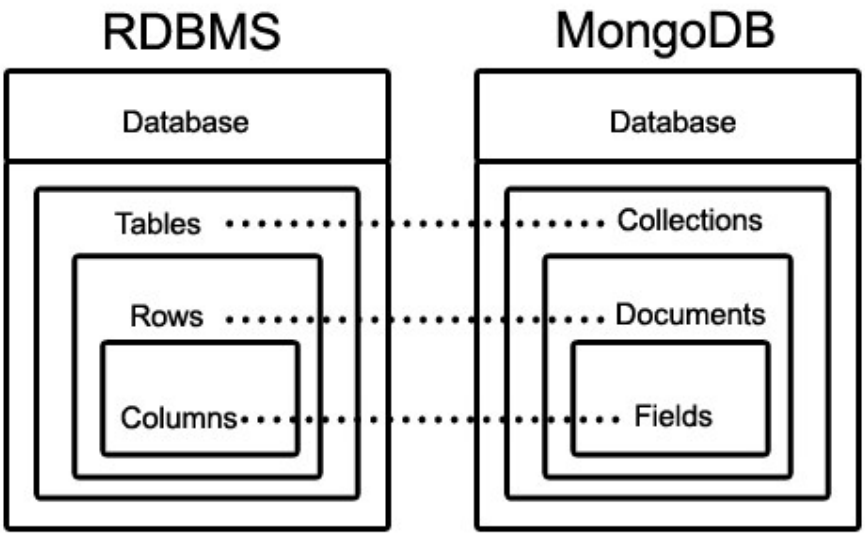
Key Differences between SQL and NoSQL

SQL	NOSQL
Relational Database management system	Distributed Database management system
Vertically Scalable	Horizontally Scalable
Fixed or predifined Schema	Dynamic Schema
Not suitable for hierarchical data storage	Best suitable for hierarchical data storage
Can be used for complex queries	Not good for complex queries

What is MongoDB?



MongoDB is an open source platform written in C++ and has a very *easy setup environment*. It is a cross-platform, document-oriented and non-structured database. MongoDB provides high performance, high availability, and auto-scaling. **Collection** and **document** are the two primarily used terms/concepts in MongoDB. Here, Collection is referred to a group of these documents, which is like an RDBMS table.



It uses the BSON format for document storage and communication with its client. BSON is a binary form of JSON.

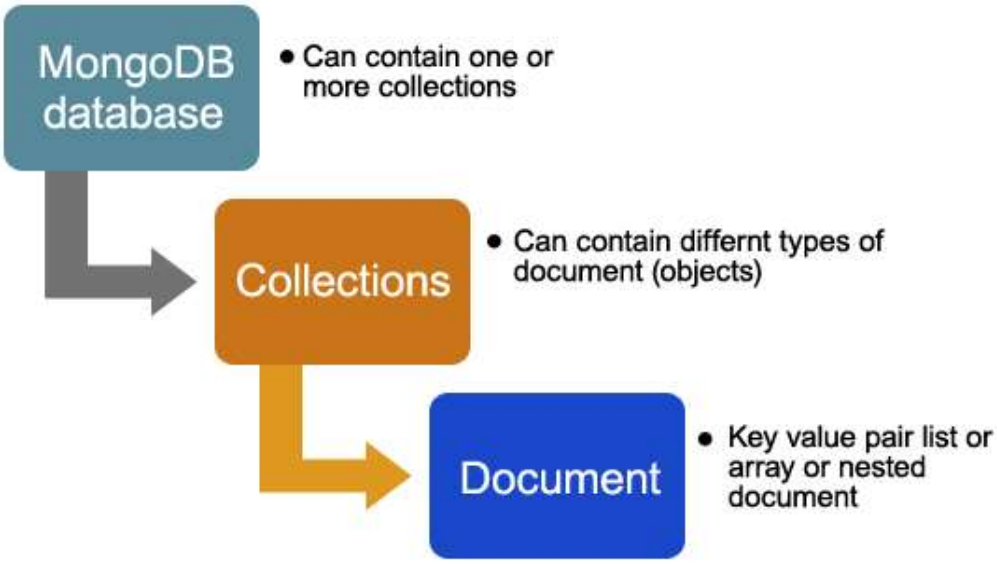
It works on an extended version of JSON kn0wn as BSON (Binary JSON), which is:

- Lightweight
- Traversable
- Efficient

These drivers are responsible for sending and receiving data in BSON format. It stores the data as a BSON Object. Encoding to BSON and Decoding to BSON again happens very quickly, and so it's so efficient. Here are a few terms related to MongoDB, which is used while using it.

Collection: Its group of MongoDB documents. This can be thought similar to a table in RDBMS like Oracle, MySQL. This collection doesn't enforce any structure. Hence schema-less MongoDB is so popular.

Document: Document is referred to as a record in MongoDB collection.



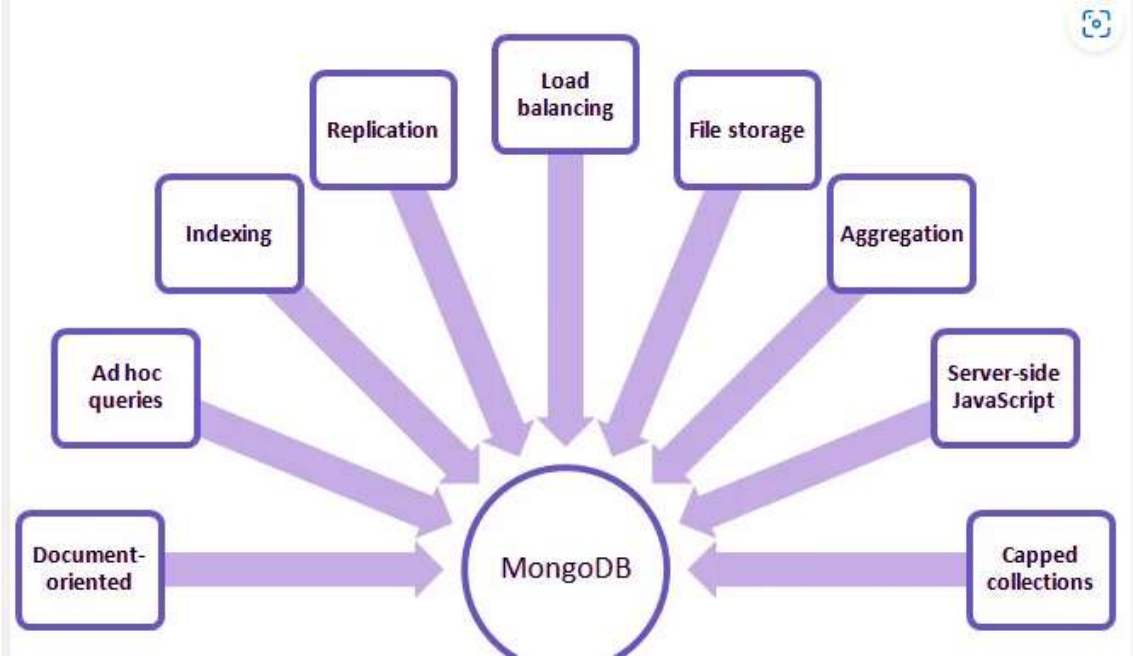
The picture shown above makes the concepts clear about collections and documents in MongoDB.

Field: It is a name-value pair in a document. A document has zero or more fields. Fields are like columns in relational databases.

Top Companies uses MongoDB

All big giants ruling the IT market nowadays relying on MongoDB. These include Adobe, Nokia, eBay, SAP, Cisco, Verizon, etc.

MongoDB Features Set



i. Ad-hoc Queries

MongoDB supports ad-hoc queries by indexing.

ii. Schema-Less Database

It is very flexible than structured databases. There is no need to type mapping.

iii Document Oriented

It is document oriented, JSON like a database.

iv. Indexing

Any document can index with primary and secondary indices.

v. Replication

It has this powerful tool. Every document has one primary node which further has two or more secondary replications.

vi. Aggregation

For efficient usability, MongoDB has aggregation framework for batch processing.

vii. GridFS

It has grid file system, so it can use to store files in multiple machines.

viii. Sharding

For the larger data sets sharding is the best feature. It distributes larger data to multiple machines.

ix. High Performance

Indexes support faster queries leading to high performance.

MongoDB Applications

the Applications of MongoDB:

- In E-commerce product catalogue.
- Big data
- Content management
- Real-time analytics and high-speed logging.
- Maintain Geolocations
- Maintaining data from social websites.

MongoDB Environment Setup | Install MongoDB on Windows

we will see how to set up the environment for MongoDB on your Windows OS. Also, we will learn how to install MongoDB.

MongoDB Environment Setup is very easy for Windows OS. To do environment setup, you will have to follow few simple steps.

- Know your Windows architecture
- Download MongoDB setup file
- Install MongoDB setup
- Set up MongoDB Environment
- Connect to the MongoDB server
- MongoDB as a Windows service
- Create configuration file
- Run MongoDB Environment setup

MongoDB Data Modeling?

- We know that MongoDB is a document-oriented database or NoSQL database.
- It is a schema-less database or we can say, it has a flexible schema.
- Unlike the structured database, where we need to determine table's schema in advance, MongoDB is very flexible in this area.
- MongoDB deals in collections, documents, and fields.

- We can have documents containing different sets of fields or structures in the same collection.
- Also, common fields in a collection can contain different types of data. This helps in easy mapping.

The key challenge in MongoDB data modeling is balancing the requirements of the application. Also, we need to assure the performance aspect effectively while modeling. Let's point out some requirements while MongoDB Data Modeling taking place.

- Design schema according to the need.
- Objects which are queried together should be contained in one document.
- Consider the frequent use cases.
- Do complex aggregation in the schema.

Considerations for MongoDB Data Modeling

Some special consideration should give while designing a data model of MongoDB. This is for high per performance scalable and efficient database. The following aspects should consider for MongoDB Data Modeling.

a. Data Usage

While designing a data model, one must consider that how applications will access the database. Also, what will be the pattern of data, such as reading, writing, updating, and deletion of data. Some applications are read centric and some are write-centric.

There are possibilities that some data use frequently whereas some data is completely static. We should consider these patterns while designing the schema.

b. Document growth

Some updates increase the size of the documents. During initialization, MongoDB assigns a fixed document size. While using embedded documents, we must analyze if the sub object can grow further out of bounds.

Otherwise, there may occur performance degradation when the size of the document crosses its limit. MongoDB relocates the document on disk if the document size exceeds the allocated space for that document.

c. Atomicity

Atomicity in contrast to the database means operations must fail or succeed as a single unit. If a parent transaction has many sub-operations, it will fail even if a single operation fails. Operations in MongoDB happen at the document level.

No single write operation can affect more than one collection. Even if it tries to affect multiple collections, these will treat as separate operations. A single write operation can insert or update the data for an entity. Hence, this facilitates atomic write operations.

However, schemas that provide atomicity in write operations may limit the applications to use the data. It may also limit the ways to modify applications. This consideration describes the challenge that comes in a way of data modeling for flexibility.

MongoDB Create Collection | MongoDB Drop Collection?

MongoDB Create Collection-

When we talk about relational databases there are tables. But in MongoDB, there are collections instead. In **MongoDB**, collections are created automatically when we refer it in any command. MongoDB will create it automatically if it doesn't exist already.

How does Collection Methods work in MongoDB?

1. We first need a MongoDB database to create a collection.

```
USE<"database name">
```

2. Mention Collection name and insert a record into it.

```
collection_name.create(<document>)
```

3. Collection Gets Created.

4. We can view any number of collections with command.

```
show collections
```

Syntax

1. To create a collection

```
db.collection_name(field:value)
```

2. To delete collection

```
collection_name.drop()
```

Example of Create Collection in MongoDB

Examples to Implement MongoDB Collection

Below are the examples of implementing MongoDB Collection:

A) Create a Collection

1. We have created a database named hospital.

Code:

```
use hospital
```

Output:

```
>>> use hospital
switched to db hospital
```

2. We have created a collection named the patient.

Code:

```
db.patient.insert({})
```

Output:

```
>>> db.patient.insert({})
WriteResult({ "nInserted" : 1 })
```

- If we don't enter any value inside (), then an error will appear. Error: no object passed to insert!

3. We have inserted a record for the patient with field: value pair.

Code:

```
db.patient.insert({name:'akash',age:23,reason:'fever'})
```

- Be careful while creating record because if we don't use (' ') for the string, then an error will appear.

```
>>> db.patient.insert({name:'akash',age:23,reason:'fever'})
WriteResult({ "nInserted" : 1 })
```

- "nInserted":1 means record inserted successfully.

4. To show the list of collections.

Code:

```
show collections
```

Output:

```
>>> show collections
patient
```

5. We have inserted multiple records into the patient collection.

Code:

```
db.patient.insert({name:'megha',age:24,reason:'fever'},
{name:'rohit',age:25,reason:'cuff'})
```

MongoDB Drop Collection –

B) Delete Collection (drop the collection)

We can delete the collection from the database. Select the database from which you want to delete the collection.

- USE <database name>
- Verify if the collection is present or not with <show collections>.
- Apply drop command with a collection.

1. We have used the hospital database.

Code:

```
use hospital
```

Output:

```
>>> use hospital
switched to db hospital
```

Code:

```
db.patienthistory.drop()
```

```
>>> db.patienthistory.drop()
true
```

- We get the true message which means collection deleted successfully.
- If the false message is displayed, then the collection is not deleted yet.

2. We have checked the collection list. We can observe that patient history is not present.

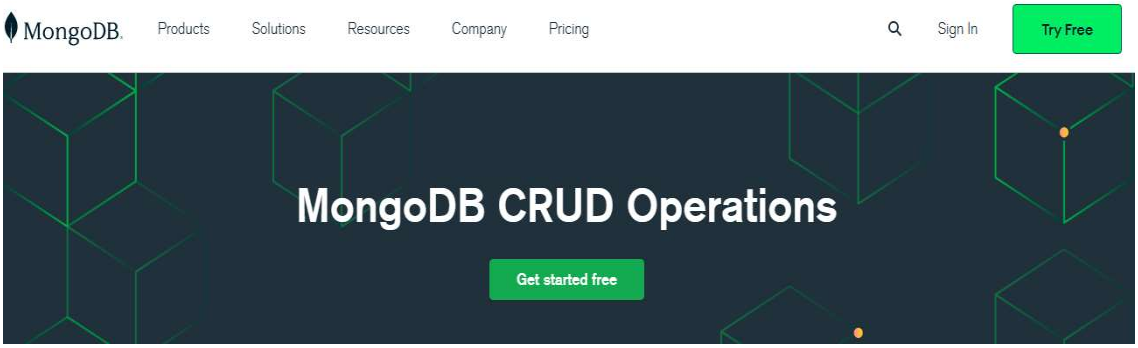
Code:

```
show collections
```

Output:

```
>>> show collections
patient
```

CRUD Operations in MongoDB?



Regardless of why you are using a MongoDB server, you'll need to perform CRUD operations on it. The basic methods of interacting with a MongoDB server are called CRUD operations. CRUD stands for Create, Read, Update, and Delete. These CRUD methods are the primary ways you will manage the data in your databases.

What is CRUD in MongoDB?

CRUD operations describe the conventions of a user-interface that let users view, search, and modify parts of the database.

MongoDB documents are modified by connecting to a server, querying the proper documents, and then changing the setting properties before sending the data back to the database to be updated. CRUD is data-oriented, and it's standardized according to HTTP action verbs.

When it comes to the individual CRUD operations:

- The Create operation is used to insert new documents in the MongoDB database.
- The Read operation is used to query a document in the database.
- The Update operation is used to modify existing documents in the database.
- The Delete operation is used to remove documents in the database.

How to Perform CRUD Operations

Now that we've defined MongoDB CRUD operations, we can take a look at how to carry out the individual operations and manipulate documents in a MongoDB database. Let's go into the processes of creating, reading, updating, and deleting documents, looking at each operation in turn.

Create Operations

For MongoDB CRUD, if the specified collection doesn't exist, the create operation will create the collection when it's executed. Create operations in MongoDB target a single collection, not multiple collections. Insert operations in MongoDB are atomic on a single document level.

MongoDB provides two different create operations that you can use to insert documents into a collection:

- db.collection.insertOne()
- db.collection.insertMany()

insertOne()

As the namesake, insertOne() allows you to insert one document into the collection. For this example, we're going to work with a collection called RecordsDB. We can insert a single entry into our collection by calling the insertOne() method on RecordsDB. We then provide the information we want to insert in the form of key-value pairs, establishing the schema.

```
db.RecordsDB.insertOne({
    name: "Marsh",
    age: "6 years",
    species: "Dog",
    ownerAddress: "380 W. Fir Ave",
    chipped: true
})
```

If the create operation is successful, a new document is created. The function will return an object where "acknowledged" is "true" and "insertID" is the newly created "ObjectId."

```
> db.RecordsDB.insertOne({
... name: "Marsh",
... age: "6 years",
... species: "Dog",
... ownerAddress: "380 W. Fir Ave",
... chipped: true
... })
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5fd989674e6b9ceb8665c57d")
}
```

insertMany()

It's possible to insert multiple items at one time by calling the *insertMany()* method on the desired collection. In this case, we pass multiple items into our chosen collection (*RecordsDB*) and separate them by commas. Within the parentheses, we use brackets to indicate that we are passing in a list of multiple entries. This is commonly referred to as a nested method.

```
db.RecordsDB.insertMany([{
    name: "Marsh",
    age: "6 years",
    species: "Dog",
    ownerAddress: "380 W. Fir Ave",
    chipped: true},
      {name: "Kitana",
      age: "4 years",
      species: "Cat",
      ownerAddress: "521 E. Cortland",
      chipped: true}])
```

```
db.RecordsDB.insertMany([{ name: "Marsh", age: "6 years", species: "Dog",
ownerAddress: "380 W. Fir Ave", chipped: true}, {name: "Kitana", age: "4 years",
species: "Cat", ownerAddress: "521 E. Cortland", chipped: true}])
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("5fd98ea9ce6e8850d88270b4"),
                ObjectId("5fd98ea9ce6e8850d88270b5")
        ]
}
```

Read Operations

The read operations allow you to supply special query filters and criteria that let you specify which documents you want. The MongoDB documentation contains more information on the available query filters. Query modifiers may also be used to change how many results are returned.

MongoDB has two methods of reading documents from a collection:

- db.collection.find()
- db.collection.findOne()

find()

In order to get all the documents from a collection, we can simply use the *find()* method on our chosen collection. Executing just the *find()* method with no arguments will return all records currently in the collection.

```
db.RecordsDB.find()
```

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years", "specie
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years", "species
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "3 years", "species"
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" : "8 years", "species
```

Here we can see that every record has an assigned “ObjectId” mapped to the “_id” key.

If you want to get more specific with a read operation and find a desired subsection of the records, you can use the previously mentioned filtering criteria to choose what results should be returned. One of the most common ways of filtering the results is to search by value.

```
db.RecordsDB.find({"species":"Cat"})
```

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years", "specie
```

findOne()

In order to get one document that satisfies the search criteria, we can simply use the *findOne()* method on our chosen collection. If multiple documents satisfy the query, this method returns the first document according to the natural order which reflects the order of documents on the disk. If no documents satisfy the search criteria, the function returns null. The function takes the following form of syntax.

```
db.{collection}.findOne({query}, {projection})
```

Let's take the following collection—say, *RecordsDB*, as an example.

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "8 years", "specie
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years", "species
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "3 years", "species"
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" : "8 years", "species
```

And, we run the following line of code:

```
db.RecordsDB.find({"age":"8 years"})
```

We would get the following result:

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "8 years", "specie
```

Notice that even though two documents meet the search criteria, only the first document that matches the search condition is returned.

Update Operations

Like create operations, update operations operate on a single collection, and they are atomic at a single document level. An update operation takes filters and criteria to select the documents you want to update. You should be careful when updating documents, as updates are permanent and can't be rolled back. This applies to delete operations as well. For MongoDB CRUD, there are three different methods of updating documents:

- db.collection.updateOne()
- db.collection.updateMany()
- db.collection.replaceOne()

updateOne()

We can update a currently existing record and change a single document with an update operation. To do this, we use the *updateOne()* method on a chosen collection, which here is “RecordsDB.” To update a document, we provide the method with two arguments: an update filter and an update action.

The update filter defines which items we want to update, and the update action defines how to update those items. We first pass in the update filter. Then, we use the “$set” key and provide the fields we want to update as a value. This method will update the first record that matches the provided filter.

```
db.RecordsDB.updateOne({name: "Marsh"}, {$set:{ownerAddress: "451 W. Coffee St. A204"}})
```

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

```
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years", "species
```

updateMany()

updateMany() allows us to update multiple items by passing in a list of items, just as we did when inserting multiple items. This update operation uses the same syntax for updating a single document.

```
db.RecordsDB.updateMany({species:"Dog"}, {$set: {age: "5"}})
```

```
{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }
```

```
> db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years", "specie
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5", "species" : "D
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5", "species" : "Dog
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" : "5", "species" : "D
```

replaceOne()

The *replaceOne()* method is used to replace a single document in the specified collection. *replaceOne()* replaces the entire document, meaning fields in the old document not contained in the new will be lost.

```
db.RecordsDB.replaceOne({name: "Kevin"}, {name: "Maki"})
```

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

```
> db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years", "specie
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5", "species" : "D
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5", "species" : "Dog
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Maki" }
```

Delete Operations

Delete operations operate on a single collection, like update and create operations. Delete operations are also atomic for a single document. You can provide delete operations with filters and criteria in order to specify which documents you would like to delete from a collection. The filter options rely on the same syntax that read operations utilize.

MongoDB has two different methods of deleting records from a collection:

- db.collection.deleteOne()
- db.collection.deleteMany()

deleteOne()

deleteOne() is used to remove a document from a specified collection on the MongoDB server. A filter criteria is used to specify the item to delete. It deletes the first record that matches the provided filter.

```
db.RecordsDB.deleteOne({name:"Maki"})
```

```
{ "acknowledged" : true, "deletedCount" : 1 }
```

```
> db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years", "specie
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5", "species" : "D
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5", "species" : "Dog
```

deleteMany()

deleteMany() is a method used to delete multiple documents from a desired collection with a single delete operation. A list is passed into the method and the individual items are defined with filter criteria as in *deleteOne()*.

```
db.RecordsDB.deleteMany({species:"Dog"})
```

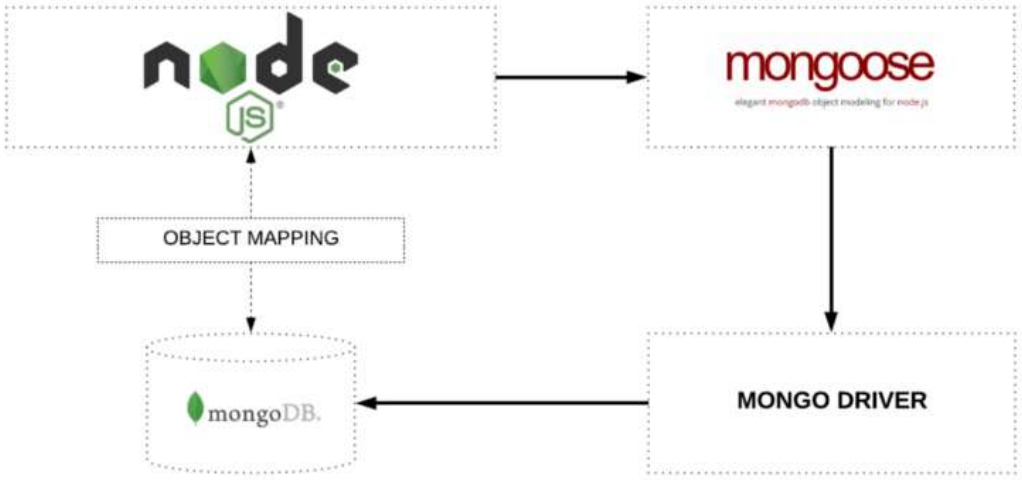
```
{ "acknowledged" : true, "deletedCount" : 2 }
```

```
> db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years", "specie
```

Introduction to Mongoose

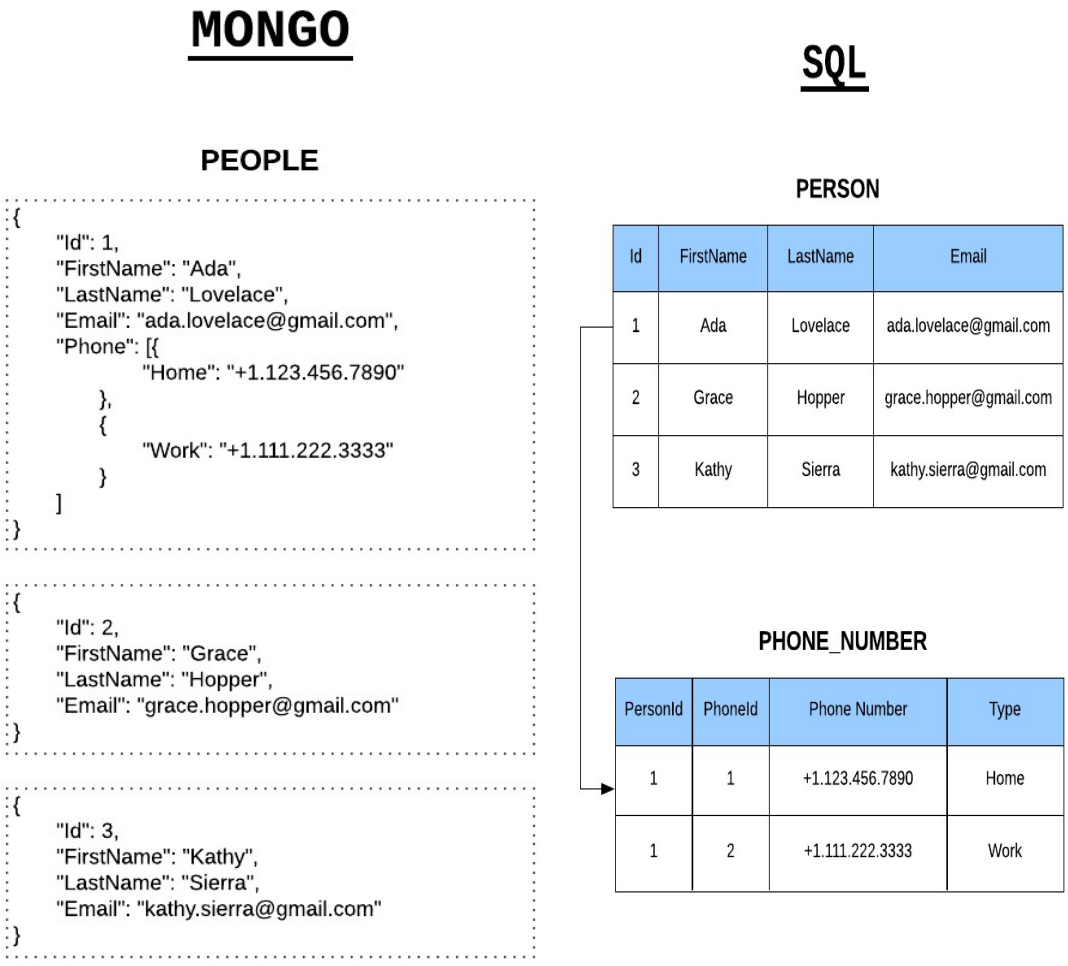
Introduction to Mongoose for MongoDB?

Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js. It manages relationships between data, provides schema validation, and is used to translate between objects in code and the representation of those objects in MongoDB.



MongoDB is a schema-less NoSQL document database. It means you can store JSON documents in it, and the structure of these documents can vary as it is not enforced like SQL databases. This is one of the advantages of using NoSQL as it speeds up application development and reduces the complexity of deployments.

Below is an example of how data is stored in Mongo vs. SQL Database:



General Terminology

- **Schemas**: Everything in Mongoose starts with a Schema. Each Schema maps to a MongoDB collection and defines the shape of the documents within that collection. It has information about properties/field types of documents. Schemas can also store information about validation and default values, and whether a particular property is required. In other words, they're blueprints for documents.
- **Model**: A model is a class with which we construct documents.

Getting Started

Simply install the `mongoose` module through npm.

```
npm install mongoose
```

Initializing Our Database

Now that Mongoose has been installed on the machine, we will use `mongoose.connect` to connect to our database, like so:

```
const mongoose = require('mongoose')
mongoose.connect('mongodb://localhost/test', {useNewUrlParser: true});
```

Here, we're connecting to a database called `test`. The first parameter is the `URI` and the second parameter is `options`.

The format of the URL is `type://username:password@host:port/database_name`.

Defining a Schema

Let's create a schema, `BookSchema` which will have a property `name` of type `String`.

```
const Schema = mongoose.Schema
const BookSchema = new Schema({
name : String
})
```

This means that the documents defined through this `BookSchema` will have one field, which is a `name` of type `String` .

Defining Models

With the `BookSchema` we just created, let's create the model `Books`.

```
const Model = mongoose.model
const Book = Model('Books',BookSchema)
```

The first parameter of `mongoose.model` is the name of our collection. The second parameter is the schema that the model will be using.

In this case, the name of our collection is `Books` which will be using the schema of `BookSchema`.