# MongoDB

## The Complete Developer's Guide

# What is MondoDB?

MongoBD is a database which is created by the company who is also called MongoDB. The name stems from the word “humongous”.  This database is built to store a lot of data but also being able to work with the huge data efficiently. Ultimately, this is a database solution.

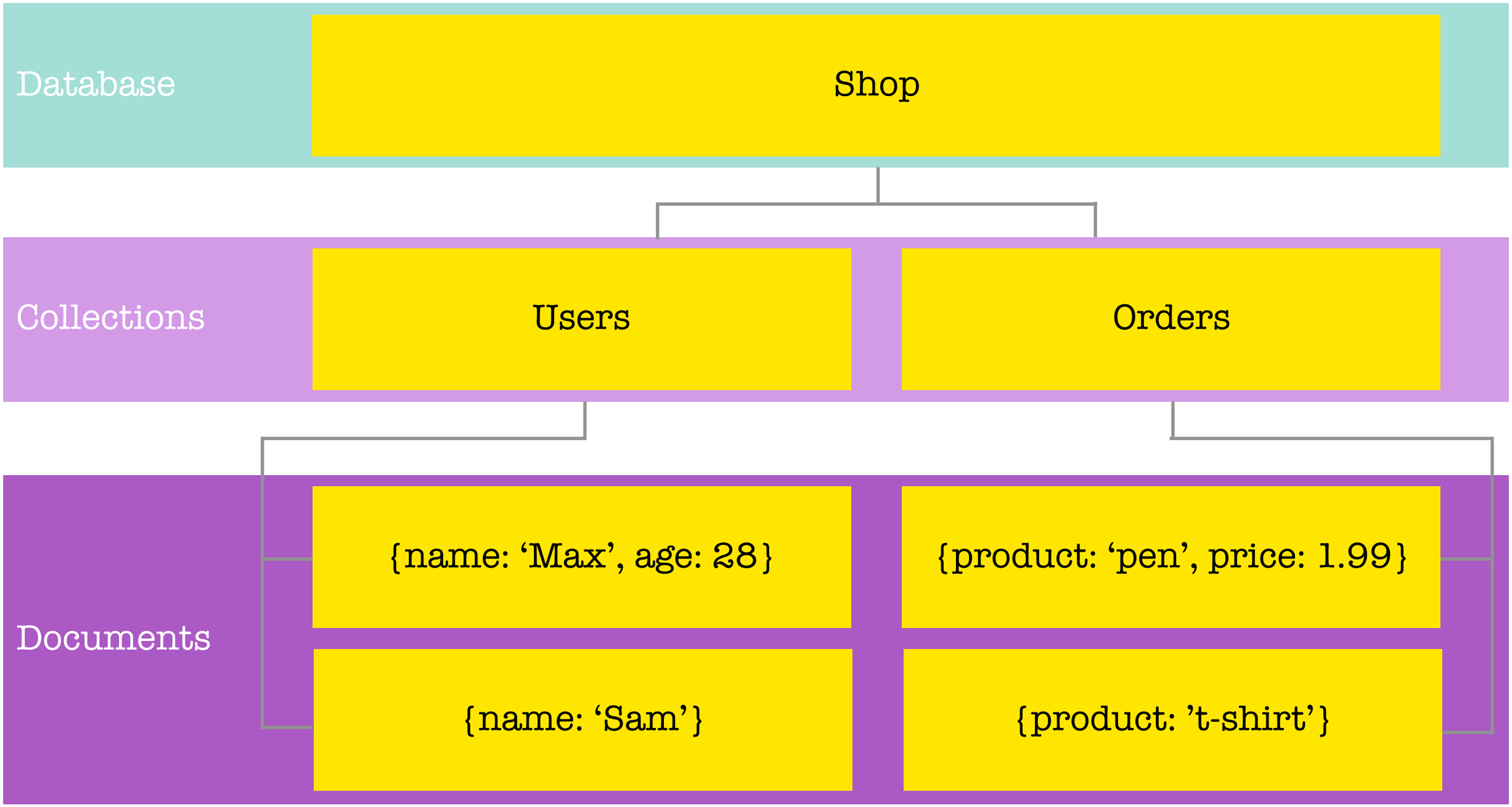
There are many database solutions such as mySQL, PostgresSQL, TSQL etc.

MongoDB is most importantly a database server that allows us to run different databases on it for example a Shop database. Within the database we would have different collections such as a Users collection or a Orders collection. We can have multiple databases and multiple collections per database.

Inside of the collection we have something called documents. Documents look like JavaScript JSON objects. Inside of a collection the documents are schema-less and can contain different data. This is the flexibility that MongoDB provides us with whereas SQL based database are very strict about the data stored within the database tables. Therefore, the MongoDB database can grow with the application needs. MongoDB is a No-SQL database.

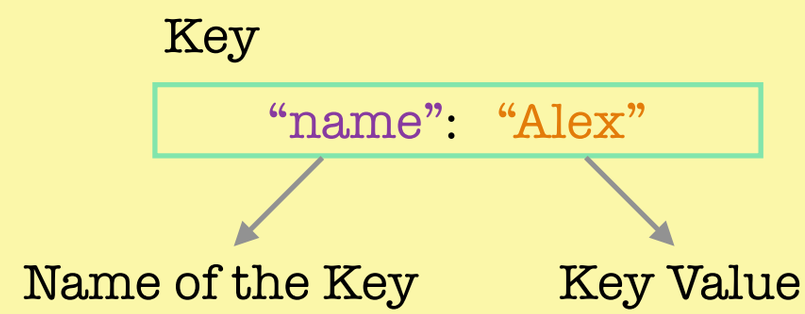
Typically we will need some kind of structure in a collection because applications typically requires some type of structure to work with the data.

## Diagram 1.1:



## JSON (BSON) Data Format:

```
{
    “name”: “Alex”,
    “age”: 29,
    “address”: {
        “city”: “Munich”
    },
    “hobbies”: [
        { “name”: “Cooking” },
        { “name”: “Football” }
    ]
}
```




The above is an example of the JSON data format. A single document is surrounded by curly brackets. The data is normally structured with a Keys. Keys consist of a Name of the Key and a Key value. The Name of the Key (*which will be referred to as Key from now on*) and the Key Value must be wrapped around quotation marks (unless if the data is a type of number).

There are different types of values we can store such as: string, number, booleans and arrays.

We can also nest documents within documents. This allows us to create complex relations between

data and store them within one document, which makes working with the data and fetching data more efficient because it is contained in one document in a logical way. SQL in contrast requires more complex method of fetching data which require joins to find data in table A and data in table B to retrieve the relevant data.

Behind the scenes on the server, MongoDB converts the JSON data to a binary version of the data which can be stored and queried more efficiently. We do not need to concern ourselves with BSON as we would tend to work with JSON data.

The whole theme of MongoDB is flexibility, optimisation and usability and it is what really sets MongoDB apart from other database solutions because it is so efficient from a performance perspective as we can query data in the format we need it instead of running complex restructuring on the server.

## The Key MongoDB Characteristics.

MongoDB is a no SQL solution because it is following an opposite concept/philosophy to SQL based databases. Instead of normalising the data i.e. storing data distributed across multiple tables where every table has a clear schema and then using relations, MongoDB goes for storing data together in a document. It does not force a schema hence schema-less/No-SQL.

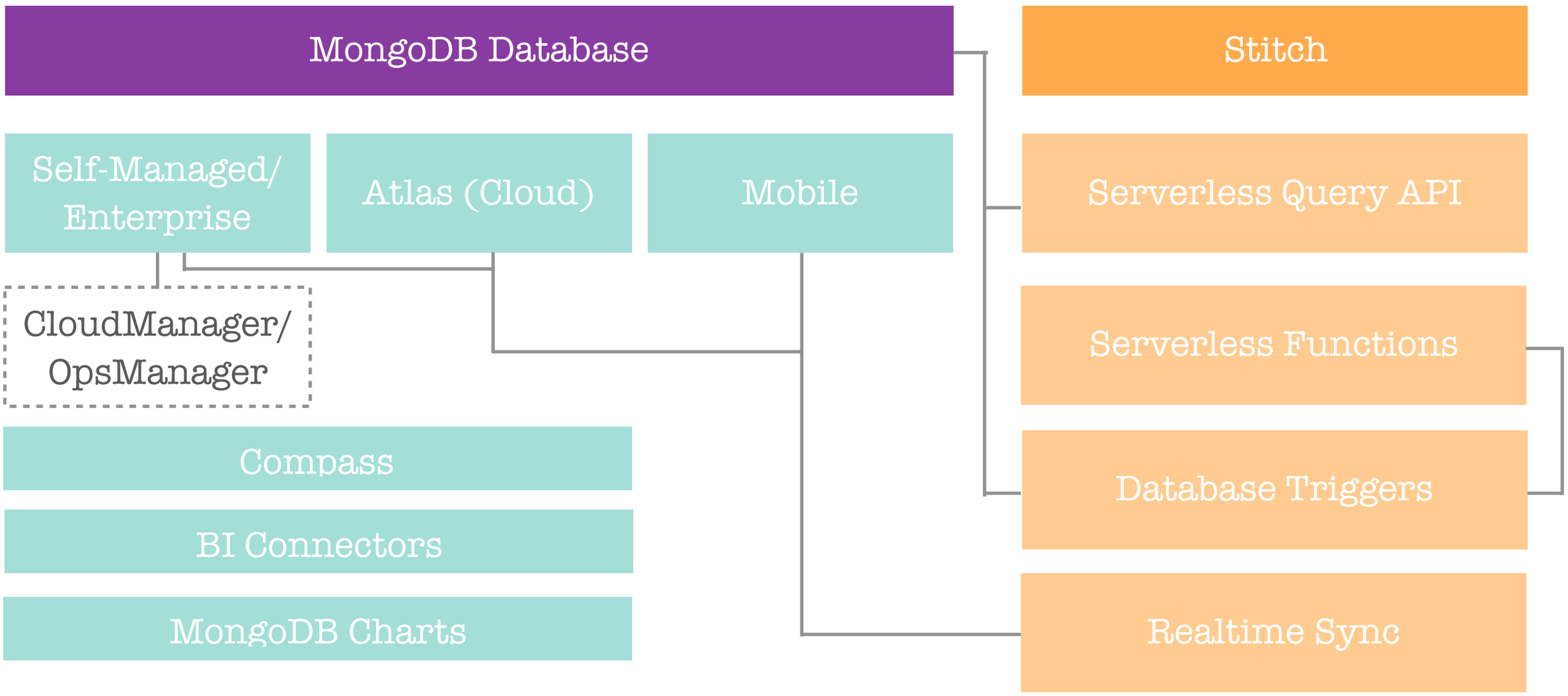
We can have multiple documents in a single collection and they can have different structures as we have seen in Diagram 1.1. This is important, it can lead to messy data but it still our responsibility as developers to work with clean data and to implement a solution that works. On the other hand this provides us with a lot of flexibility. We could use mongoDB for applications that might still evolve, where the exact data requirements are not set yet. MongoDB allows us to started and we could always add data with more information in the same collection at a later point in time.

We also work with less relations. There are some relations, but with these embedded (nested) documents, we have less collections (tables) which we connect but instead we store data together. This is where the efficiencies is derived from, since data is stored together and when we fetch data from our application it does not require to reach out to multiple tables and merge the data because all the data is already within the single collection. This is where the speed, performance and flexibility comes from and can be seen beneficial for when building applications. This is the main reason why No-SQL solutions are so popular for read and write heavy applications.

# MongoDB Ecosystem

The below Diagram 1.2 is the current snapshot of the MongoDB companies ecosystem and product offerings. The focus of this guide is on the MongoDB database used locally on our machines and on the cloud using Atlas. We will also dive in Compass and the Stitch world of MongoDB.

## Diagram 1.2:



# Installing MongoDB

MongoBD runs on all Operating Systems (OS) which include Windows/Mac/Linux. To install MongoDB we can visit their webpage on:

https://www.mongodb.com/

Under products select MongoDB server and download the MongoDB Community Server for our OS platform of choice. Install the MongoDB Server by following the installation steps.

**Important Note**: On Windows when installing click on the Custom Setup Type. MongoDB will be installed as a service which will be slightly different to how MongoDB runs on Mac & Linux.

On Mac and Linux we simply have a extracted folder which contains files. We would copy all the contents within this folder and paste them into any place within our OS i.e wherever we would want to install MongoDB.

We would then want to create a folder called data and a sub-folder called db anywhere within our OS, preferably in the root of the OS.

On Windows open up the command prompt or on Mac/Linux open up the terminal. This is where

we are going to spend most of out time using special commands and queries.  Run the following command:

```
$ mongo
```

This should return command not found.

To fix this problem on a Mac go to the user folder and find a file called .bash_profile file (*if this does not exist we could simply create it*). Edit the file using a text editor. Add the following line:

```
export PATH=/Users/Username/mongobd/bin:$PATH
```

The path should be wherever we placed the mongoDB binary zip files. We need to add :$PATH at the end on Mac/Linux to make sure all our other commands work on our OS. Save the file and close the file.

**Important Note:** if you run into a problem on not being able to edit the .bash_profile using text editor use the following command to edit it within the terminal:

```
$ sudo nano ~/ .bash_profile
```

This will allow you to edit the file within the terminal and enter the mongo bin file path. Press CRTL + o to save and CTRL + x to exit the nano edit.

To fix this on a Windows OS, we need to create an environment variable. Press the windows key and type environment which should suggest the Edit Environment Variable option. Under the user variables edit Path to add the directory path to where we installed the mongoDB files:

C:\Program Files\MongoDB\Server\4.0\bin

Restart the terminal/command prompt and now run the command:

$ mongo

This should now return a error of connect failed on Mac/Linx.

On Windows it will connect because MongoDB is running as a service and has already started as a background service because we would have checked this during the installation. If we open the command prompt as administrator and ran the command 'net stop MongoDB' this will stop the background service running automatically and we can manually start and stop the MongoDB service running on windows. DO NOT RUN THIS COMMAND ON MAC/LINUX.

The mongo command is the client which allows us to connect to the server and then run commands on the databases. To start the service on Mac/Linux we would use the following command:

$ mongod

When running this command to start the server it may fail if we chose a different default /data/db folder. If we used a different folder and not within the root of our OS we would need to start the mongod command instance followed by the --dbpath flag and the place where the /data/db is located if not within the root directory.

```
$ sudo mongod –dbpath "/data/db"
```

On Mac we would need to run the mongod command every time we wish to run the mongoDB service whereas on Windows this will run automatically even after restarting the system.

Now that we have the mongod server running minimise the terminal on Mac/Linux and open up a new terminal. We cannot close the mongod server terminal because it is running the service and if closed the mongoDB server everything will stop working and we cannot continue to work with the database server. Pressing the CTRL + C keys within the terminal will quit the mongod service, but we would need to re-run the mongod command again should we wish to run the server again.

We are now in the mongo shell which is the environment where we can run commands against our database server. We can create new databases, collections and documents which we will now focus on in the following sections.

# Time to get Started

Now that we have the mongod server running and we can now connect to it using the mongo shell we can now enter the following basic commands in the mongo terminal:

| Command | Description |
|---|---|
| $ cls | Clear the terminal. |
| $ show dbs | Display existing databases (there are three default databases: admin, config and local which store meta data). |
| $ use databaseName | Connect/Switch to a database. If the database does not exist it will implicitly create a new database using the databaseName. It will not create the database until a collection and document is added. |
| $ db.collectionName.insertOne( {"name of key": "key value"} ) | Create a new collection. The db relates to the current connected database. This will implicitly create a new collection if it does not exist. We must pass at least one new data in the collection using the .insert() command passing in a JSON object. This will return the object to confirm the data was inserted into the database. |

**Important Note:** we can omit the quotes around the name of the key within the shell but we must contain the quotes for the key value unless the key value is type of number. This is a feature within the mongo shell which work behind the scenes. MongoDB will also generate a uniqueId for new documents inserted into the collection.

| Command | Description |
|---|---|
| $ db.collectionName.find() | Display the document within the database collection. |
| $ db.collectionName.find().pretty() | Display the documents within the database collection but prettify the data in a more humanly readable format. |

This is a very basic introductory look at the following shell commands we can run in the mongo terminal to create a new database, switch to a database, create a collection and documents and display all the documents within a database collection either in the standard or pretty format.

**Tip**: to run the mongod server on a different port to the default port 27017 by run the following command. Note you would need to specify the port when running the mongo shell command as well. You would use this in case the default port is being used by something else.

```
$ sudo mongo --port 27018
$ mongo --port 27018
```

# Shell vs Drivers

The shell is a great neutral ground for working with mongoDB. Drivers are packages we install for different programming languages the application might be written in. There are a whole host of drivers for the various application server languages such as PHP, node, C#, python etc. Drivers are the bridges  between the programming language and the mongoDB server.

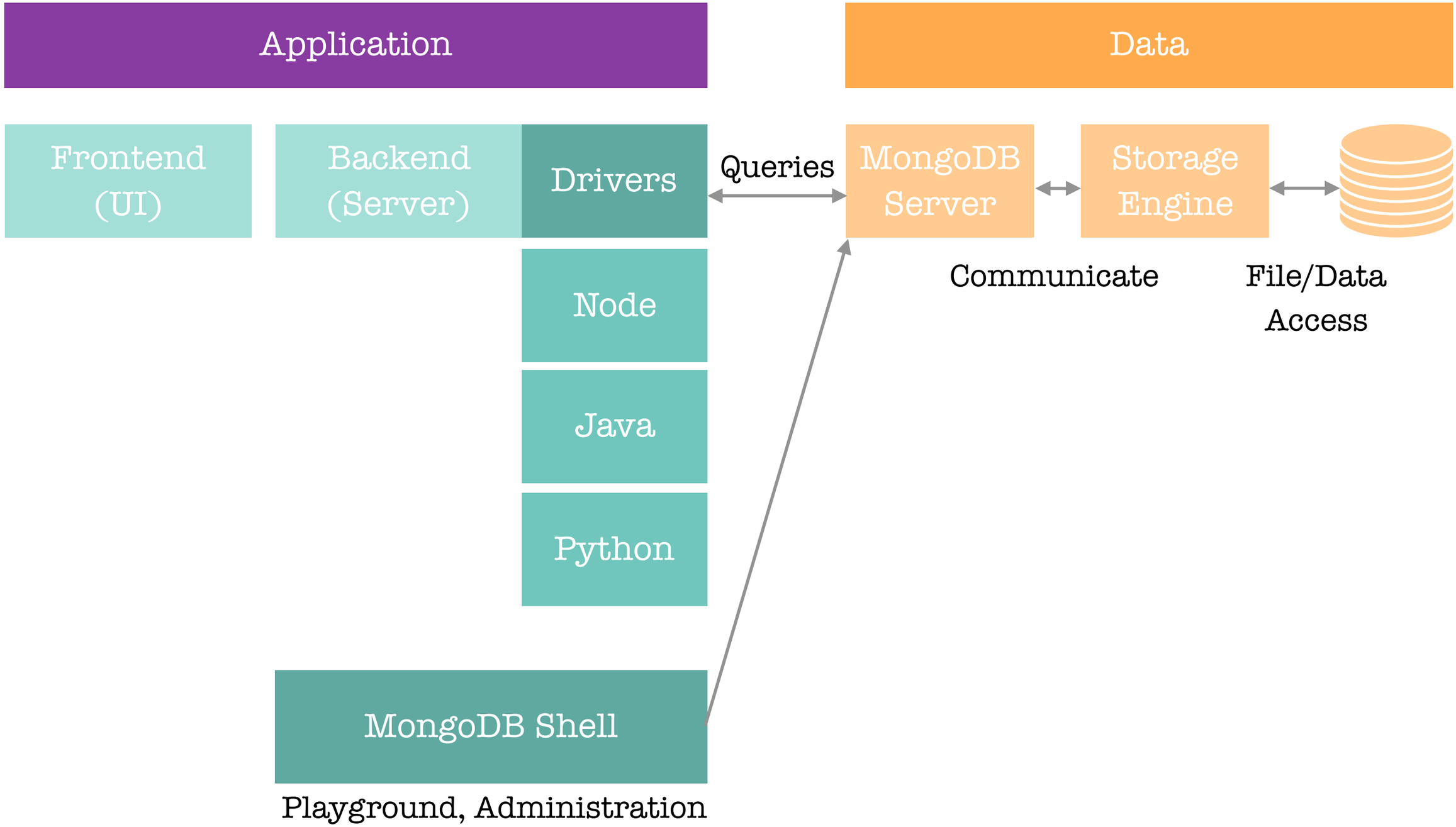
As it turns out, in these drivers, we would use the same command as we use in the shell, they are just slightly adjusted to the syntax of the language we are working with.

The drivers can be found on the mongoDB website:
https://docs.mongodb.com/ecosystem/drivers/

Throughout this document we will continue to use the Shell commands as it is the neutral commands. We can take the knowledge of how to insert, configure inserts, query data, filter data, sort data and many more shell commands. These commands will continue to work when we use the drivers but we would need to make reference to the driver documentation to understand how to use the shell commands but using the programming language syntax to perform the commands using the drivers. This will make us more flexible with the language we use when building applications that uses mongoDB.

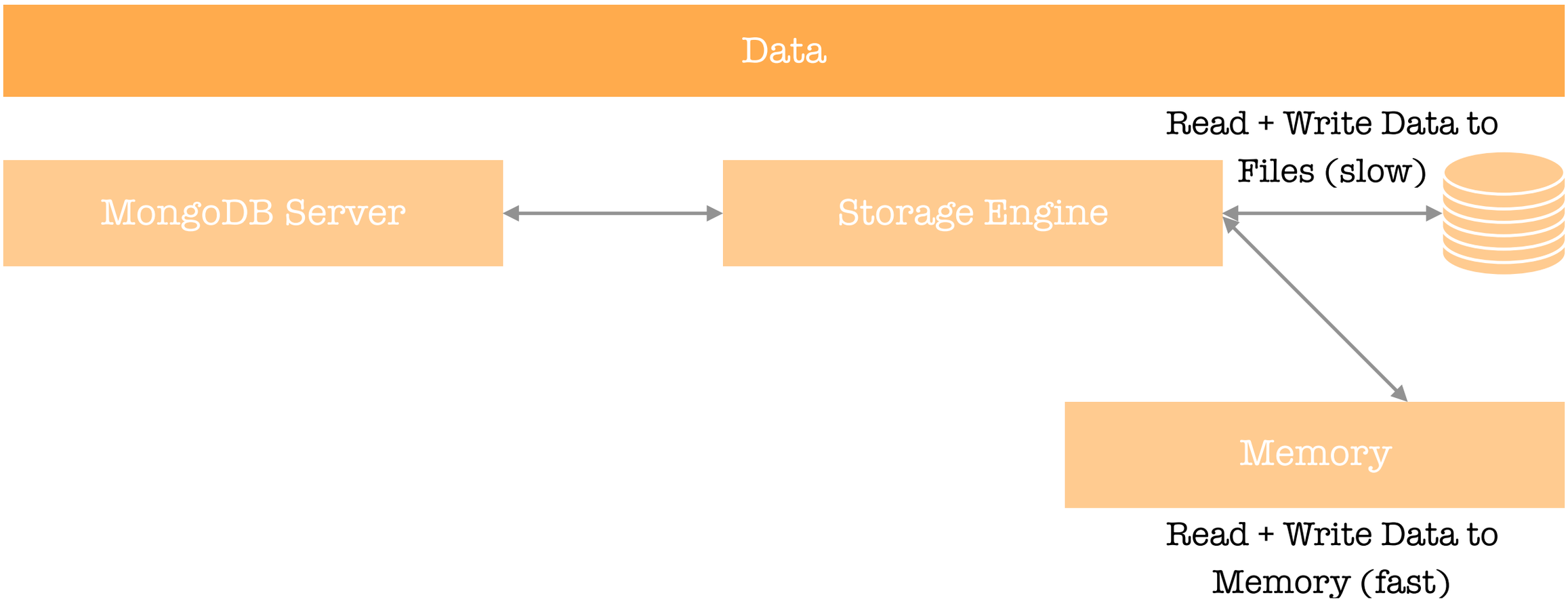
# MongoDB & Clients: The Big Picture

## Diagram 1.3:



# MongoDB & Clients: The Big Picture

## Diagram 1.4:




As we can see in Diagram 1.3 the application driver/shell communicates to the mongoDB server. The MongoDB server communicates with the storage engine. It is the Storage Engine which deals with the data passed along by the MongoDB Server, and as Diagram 1.4 depicts it will read/write to database and/or memory.

# Create, Read, Update & Delete (CRUD)

We could use MongoDB to create a variety of things such as an application, Analytics/BI Tools or data administration. In an application case, we may have an app where the user interacts with our code (the code can be written in any programming language) and the mongoDB driver will be included in the application. In the case of a Analytics/BI Tools we may use the BI Connector/Shell provided by mongoDB or another import mechanism provided by our BI tool. Finally, in the database administrator case we would interact with the mongoDB shell.

In all the above cases we would want to interact with the mongoDB server. In an application we would typically want to be able to create, read, update or delete elements e.g. a blog post app. With analytics, we would at least want to be able to read the data and as an admins we would probably want to do all the CRUD actions.

CRUD are the only actions we would want to perform with our data i.e. to create it, manage it or read it. We perform all these actions using the mongoDB server.

## Diagram 1.5:

| CREATE | UPDATE |
| --- | --- |
| insertOne(data, options)<br>insertMany(data, options) | updateOne(filter, data, options)<br>updateMany(filter, data, options)<br>replaceOne(filter, data, options) |
| **READ** | **DELETE** |
| findOne(filter, options)<br>find(filter, options) | deleteOne(filter, options)<br>deleteMany(filter, options) |

The above are the four CRUD operations and the commands we can run for each action. In later sections we will focus on each CRUD action individually to understand in-depth each of the actions and syntax/command we can use when performing CRUD operation with our mongoDB data collection and documents.

# Finding, Inserting, Updating & Deleting Elements

To show all the existing databases within the mongoDB server we use the command “show dbs” while we use the use followed by the database name to switch to a database. The db will then relate to the switched database.

To perform any CRUD operations, these commands must always be performed/executed on a collection where you want to create/update/delete documents. Below are example snippets of CRUD commands on a fictitious flights database (where the collection is called flightData).

```
$ db.flighData.insertOne( {distance: 1200} )
```

This will add a single document to the collection as we have seen previously.

```
$ db.flightData.deleteOne( {departureAirport: “LHR”} )
$ db.flightData.deleteMany( {departureAirport: “LHR”} )
```

The delete command takes in a filter. To add a filter we would use the curly brackets passing in

which name of key and key value of the data we wish to filter an delete. In the above example we used the departureAirport key and the value of TXL. The deleteOne command will find the first document in our database collection that meets the criteria and deletes it. The command will return:

```
{ “acknowledged” : true, “deleteCount” : 1 }
```

If a document was deleted in the collection this will show the number of deleted documents (the deleteOne command will always return 1). If no documents matched the filter and none were deleted the returned deleteCount value will be 0.

The deleteMany in contrast will delete many documents at once where the documents matches the filter criteria specified.

Note: The easiest way to delete all data in a collection is to delete the collection itself.

```
$ db.flightData.updateOne( {distance: 1200}, { $set: {marker: “delete”} } )
$ db.flightData.updateMany( {distance: 1200}, { $set: {marker: “delete”} } )
```

The update command takes in 3 argument/parameters. The first is the filter which is similar to the delete command. The second is how we would want to update/change the data. We must use the

{ $set: { } } keyword (*anything with a $ dollar sign in front of the keyword is a reserved word in mongoDB*) which lets mongoDB know how we are describing the changes we want to make to a document. If the update key:value does not exist, this will create a new key:value property within the document else it will update the existing key:value with the new value passed in. The third parameter is options which we will analyse in great detail in the latter sections.

**Important Note**: when passing in a filter we can also pass in empty curly brackets { } which will select all documents within the collection.

If successful with updating many this will return within the terminal an acknowledgement as seen below, where the number of matched the filter criteria and the number of data modified:
{ "acknowledged" : true, "matchedCount" : 2, "modified" : 2 }

If we were to delete all the documents within a collection and use the command to find data in that collection i.e using the db.flightData.find().pretty() command, the terminal will return empty/nothing as there are no existing documents to read/display.

The above demonstrates how we can find, insert, update and delete elements using the update and delete command.

Now we have seen how we can use insertOne() to add a single document into our collection. However, what if we want to add more than one document? We would use the inserMany() command instead.

```
db.flightData.insertMany( [
    {
        “departureAirport”: “LHT”,
        “arrivalAirport”: “TXL”
    } ,
    {
        “departureAirport”: “MUC”,
        “arrivalAirport”: “SFO”
    }
] )
```

We pass in an array of objects in order to add multiple documents into our database collection. The square brackets is used to declare an array. The curly brackets declare a object and we must use comma’s to separate each object. If successful, this will return acknowledged of true and the insertdIds of each object/document added into the collection.

**Important Note**: mongoDB by default will create a unique id for each new document which is assigned to a name of key called "_id" followed by a random generated key. When inserting a object we could assign our own unique id using the _id key followed by a unique value. If we insert a object and pass in our own _id key value and the value is not unique this will return a duplicate key error collection in the terminal. We must always use a unique id for our documents and if we do not specify a value for _id then mongoDB will generate one for us automatically.

## Understanding the Basics & CRUD Operations

# Diving Deeper Into Finding Data

Currently we have seen the .find() function used without passing any arguments for finding data within a collection. This will retrieve all the data within the collection. Just as we would use filter to specify a particular records or documents when deleting or updating a collection, we can also filter when finding data.

We can pass a document into the find function which will be treated as a filter as seen in the example below. This allows us to retrieve a subset of the data rather than the whole data within an application.

```
db.flightData.find( {intercontinental : true } ).pretty()
```

We can also use logical queries to retrieve more than one document within a collection that matches the criteria as demonstrated in the below example. We query using another object and then one of the special operators in mongoDB.

```
db.flightData.find( {distance: {$gt: 1000 } } ).pretty()
```

In the above we are using the $gt: operator which is used for finding documents “greater than” the value specified. If we were to use the findOne() operator this will return the first record within the collection that matches the criteria.



# Update vs UpdateManay

Previously we have seen the updateOne() and updateMany() functions. However, we can also use another update function called update() as seen in the example below:

```
db.flightData.update( { _id: ObjectId(“abc123”) }, { $set: { delayed: true } } )
```

The update() function works exactly like the updateMany() function where all matching documents to the filter are updated. The difference between update() and updateMany() is that the $set:

operator is not required for the update() function whereas this will cause an error for either the updateOne() and updateMany() functions. So we can write the above syntax like so and would not get an error:

```
db.flightData.update( { _id: ObjectId("abc123") }, { delayed: true } )
```

The second and main difference is that the update function takes the new update object and replaces the existing object (*this does not affect the unique id*) updating the document. It will only patch the update object instead of replacing the whole existing object (just like the updateOne() and updateMany() functions), if we were to use the $set: operator, otherwise it would override the existing document.

This is something to be aware of when using the update() function. If we intend to replace the whole existing document with a new object then we can omit the $set: operator. In general it is recommended to use updateOne() and updateMany() to avoid this issue.

If, however, we want to replace a document we should use the replaceOne() function. Again, we would place our filter and the object we want to replace with. This is a more explicit and more safer way of replacing the data in a collection.

```
db.flightData.replaceOne( { id: ObjectId("abc123") }, { departureAirport: "LHT", distance: 950 } )
```

# Understanding Find() & The Cursor Object

If we have a passengers collection which stores the name and age of passengers and we want to retrieve all the documents within the passenger collection we can use the find() function as we have seen previously.

db.passengers.find().pretty()

**Useful Tip:** when writing commands in the shell we can use the tab key to autocomplete for example if we wrote db.passe and tab on our keyboard, this should auto-complete db.passengers.

We will notice where a collection has many data, the find() function will return all the data but display all the data with the shell. If we scroll down to the last record we should see Type "it" for more within the shell. If we type the command it and press enter, this will display more data from the returned find() function. The find() command in general returns back what is called a Cursor Object and not all of the data.

The find() does not give an array of all the documents within a collection. This makes sense as the

collection could be really large and if the find() was to return the whole array, imagine if a collection had 2million documents – this could take a really long time but also send a lot of data over the connection.

The Cursor Object is an object that has many meta data behind it that allows us to cycle through the results, which is what the "it" command did. It used the Cursor Object to fetch the next group (cycle) of data from the collection.

We can use other methods on the find() function such as toArray() which will exhaust the cursor i.e. go through all of the cursors and fetch back all the documents within the array (i.e. not stopping after the first 20 documents – a feature within the mongoDB shell).

```
db.passengers.find().toArray()
```

There is a forEach method that can also be used on the find() function. The forEach allows us to write some code to do something on every element that is in the database. The syntax can be found within the driver documents for whichever language we are using for our application e.g. PHP or JavaScript etc. Below is a JavaScript function which the shell can also use:

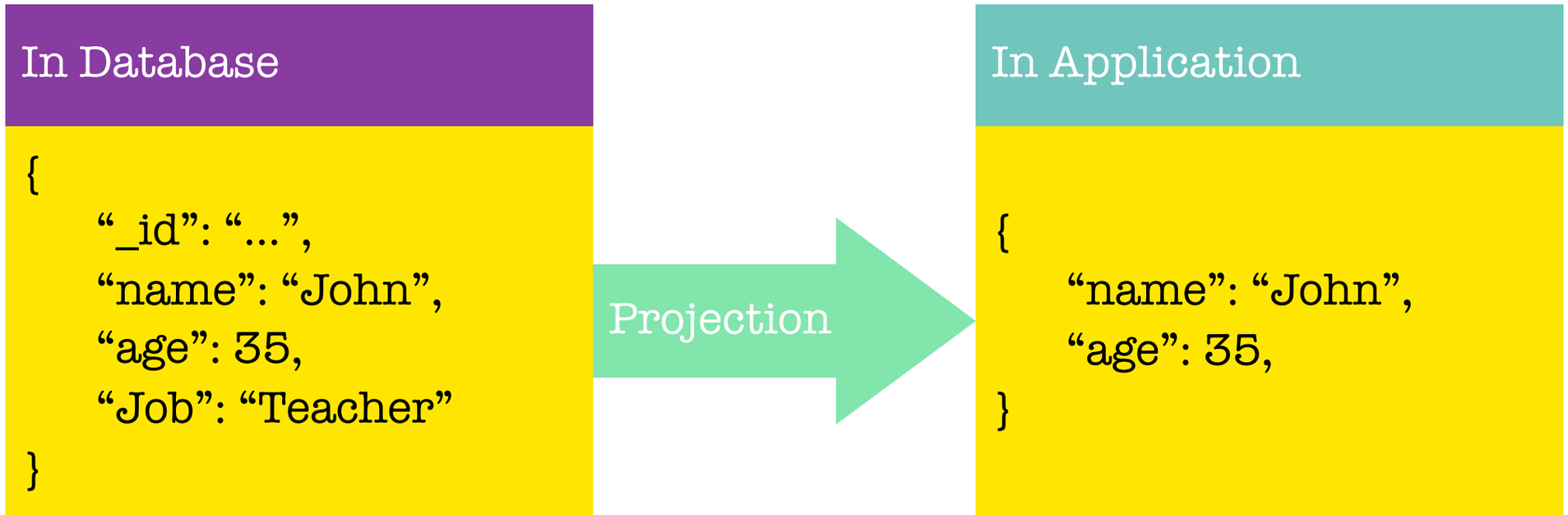
```
db.passengers.find().forEach( (document) => { printjson(document) } )
```

The forEach function in JavaScript gets the document object passed in automatically into the arrow function and we can call this whatever we want i.e. passengersData, data, x, etc. In the above we called this document. We can then use this object and do whatever we want i.e. we used the printjson() command to print/output the document data as JSON. The above will also return all the documents within the collection because the forEach loops on every Cursor Object.

To conclude, the find() function does not provide us with all the documents in a collection even though it may look like it in some circumstances where there are very little data within a collection. Instead it returns a Cursor Object which we can cycle through the return more documents from the collection. It is unto us as the developer to use the cursor to either force it to get all the documents from a collection and place it in an array or better using the forEach or other methods to retrieve more than 20 documents (*the default number of items returned in the shell*) from the collection. Note the forEach is more efficient because it fetches/returns objects on demand through each iteration rather than fetching all the data in advance and loaded into memory which saves both on bandwidth and memory.

The Cursor Object is also the reason why we cannot use the .pretty() command on the findOne() function because the findOne returns one document and not a Cursor Object. For Insert, Update and Delete commands the Cursor Object does not exist because these methods do not fetch data, they simply manipulate the data instead.

# Understanding Projections




Imagine in our database we have the data for a person record and in within our application we do not need all the data from the document but only the name and age to display on our web application. We could fetch all the data and filter/manipulate the data within our application in any programming language. However, this approach will still have an impact on the bandwidth by fetching unnecessary data – something we want to prevent. It is better to filter the data out from the mongoDB server and this is exactly what projection allows us to do.

Below are examples of using projections to filter the necessary data to retrieve from our find query.

```
db.passengers.find( {}, {name: 1} ).pretty()
```

We need to pass in a first argument to filter the find search (note: a empty object will retrieve all documents). The second argument allows us to project. A projection is setup by passing another document but specifying which key:value pairs we want to retrieve back. The one means to include it in the data returned to us.

The above will return all the passengers document but only the name and id, omitting the age from the returned search results. The id is a special field in our data and by default it is always included. To exclude the id from the returned results, we must explicitly exclude it. To exclude something explicitly we would specify the name of key and set the value to zero as seen below:

```
db.passengers.find( {}, {name: 1, _id:0} ).pretty()
```

Note: we could do the same for age (e.g. age: 0), however, this is not required because the default is everything but the _id is not included in the projection unless explicitly specified using the one.

The data transformation/filtering is occurring on the mongoDB server before the data is shipped to us and is something that we would want because we do not want to retrieve unnecessary data which will impact on the bandwidth.

# Embedded Documents & Arrays

Embedded documents is a core feature of mongoDB. Embedded documents allows us to nest other documents within each other and having one overarching document in the collection.

There are two hard limits to nesting/embedded documents:

1. We can have up to 100 level of nesting (a hard limit) in mongoDB.
2. The overall document size has to be below 16mb

The size limit for documents may seem small but since we are only storing text and not files (we would use file storage for files), 16mb is more than enough.

Along with embedded documents, another documents we can store are arrays and this is not strictly linked to embedded documents, we can have arrays of embedded documents, but arrays can hold any data. This means we have list of data in a document.

Below are examples of embedded documents and arrays.

```
db.flightData.updateMany( {}, {$set: {status: {description: "on-time", lastUpdated: "1 hour ago"} } } )
```

In the above example we have added a new document property called status which has a embedded/nested document of description and lastUpdated. If we output the document using .find() function, the below document would now look something like the below:

```
{
    "_id": …
    "departure": "LHT",
    "arrivalAirport": "TXL",
    "status": {
        "description": "on-time",
        "lastUpdated" "1 hour ago",
    }
}
```

Note: we could add more nested child documents i.e. description could have a child nested document called details and that child could have further nested child documents and so on.

```
db.passengers.updateOne( {name: "Albert Twostone"}, {$set: {hobbies: ["Cooking", "Reading"] } } )
```

Arrays are marked with square brackets. Inside the array we can have any data, this could be multiple documents (i.e. using the curly brackets {}), numbers, strings, booleans etc.

If we were to output the document using the .find() function, the document would look like something below:

```
{
    "_id": ...,
    "name": "Albert Twostone",
    "age": 63,
    "hobbies": [
        "Cooking",
        "Reading"
    ]
}
```

Albert Twostone will be the only person with hobbies and this will be a list of data. It is important to note that hobbies is not a nested/embedded document but simply a list of data.

# Accessing Structured Data

To access structured data within a document we could use the following syntax:

db.passengers.findOne( {name: “Albert Twostone”} ).hobbies

We can specify the name of a structured data within a document by using the find query and then using the name of key we wish to access from the document, in the above we wanted to access the hobbies data which will return the hobbies array as the output:

[“Cooking”, “Reading”]

We can also search for all documents that have hobbies of Cooking using the syntax below as we have seen previously. This will return the whole document entry where someone has Cooking as a hobby. MongoDB is clever enough to look in arrays to find documents that match the criteria.

db.passengers( {hobbies: “Cooking”} ).pretty()

Below is an example of searching for objects (this includes searching within nested documents):

db.flightData.find( {“status.description”: “on-time”} ).pretty()

We use the dot notation to drill into our embedded documents to query our data. It is important that we wrap the dot notation in quotations (e.g. “status.description”) otherwise the find() function would fail.

This would return all documents (the whole document) where the drilled criteria matches. This allows us to query by nested/embedded documents. We can drill as far as we need to using the dot notation as seen in the example below:

```
db.flightData.find( {“status.details.responsible”: “John Doe”} ).pretty()
```

This dot notation is a very important syntax to understand as we would use this a lot to query our data within our mongoDB database.

## Understanding the Basics & CRUD Operations

# Conclusion

We have now covered all the basic and core features of mongoDB to understand how mongoDB works and how we can work with it i.e. store, update, delete and read data within the database as well as how we can structure our data.

# Resetting The Database

To purge all the data within our mongoDB database server we would use the following command:

```
use databaseName
db.dropDatabase()
```

We must first switch to the database using the use command followed by the database name. Once we have switched to the desired database we can reference the current database using db and then call on the dropDatabase() command which will purge the specified database and its data.

Similarly, we could get rid of a single collection in a database using the following command:

```
db.myCollection.drop()
```

The myCollection should relate to the collection name.

These commands will allow us to clean our database server by removing the database/collections that we do not want to keep on our mongoDB server.

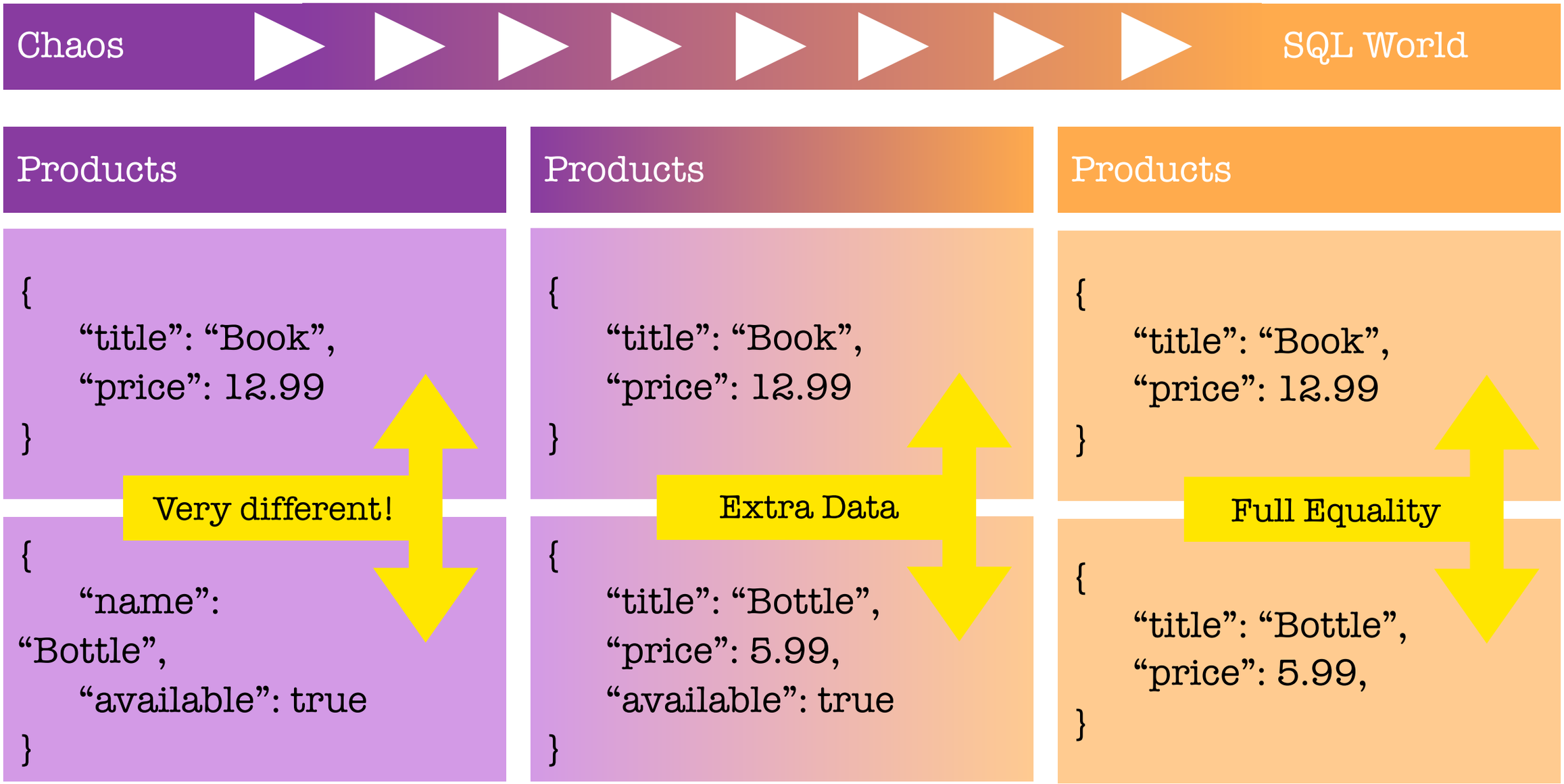
# Why Do We Use Schemas?

There is one important question to ask – wasn't mongoDB all about having **no** data Schemas i.e. Schema-less. To answer this question, mongoDB enforces no Schemas. Documents do not have to use the same schema inside of one collection. Our documents can look like whatever we want it to look like and we can have totally different documents in one and the same collection i.e. we can mix different schemas.

Schemas are the structure of one document i.e. how does it look like, which fields does it have and what types of value do these fields have. MongoDB des not enforce schemas; however, that does not mean that we cannot use some kind of schema and in reality we would indeed have some form of schema for our documents. It is in our interest if we were to build a backend database that we have some form of structure to the types of documents we are storing. This would make it easier for us to query our database and get the relevant data and then cycle through this data using a programming language to display the relevant data within our application.

We are most likely to have some form of schemas because we as developers would want it and our applications will need it. Whilst we are not forced to have a schema we would probably end up with some kind of schema structure and this is important to understand.

# Structuring Documents




We can use any of the structured approach in the diagram above depending on how we require it in our applications. In reality we would tend to use the approach in the middle or on the right.

The middle approach used the best of both words where there are some structure to the data, however, it also has the flexibility advantage that mongoDB provides us so that we can store extra information.

Note: we can assign the null value to properties in order to have a structured approach although the data may not have any actual values associated with the property. A null value is considered a valid value and therefore we can use a SQL (structured) type approach with all our documents.

There is no single best practice with how to set the structure of our data within our documents and it is up to us as developers to use the best structure that works best for our applications or whichever is to our personal preference.

### Schemas & Relations: How to Structure Documents

## Data Types

Now that we understand that we are free to define our own schemas/structure for our documents, we are now going to analyse the different data types we can use in mongoDB.

Data Types are the types of data we can save in the fields within our documents. The below table break the different data types for us:

| Type | Example Value |
| --- | --- |
| String | “John Doe” |
| Boolean | TRUE |
| NumberInt (int32) | 55, 100, 145 |
| NumberLong (int64) | 10000000000 |
| NumberDecimal | 12.99 |
| ObjectId | ObjectId(“123abc”) |
| ISODate | ISODate(“2019–02–09”) |
| Timestamp | Timestamp(11421532) |
| Embedded Documents | {“a”: {...}} |
| Arrays | {“b”: [...]} |

Notice how the text type requires quotation marks (*single or double*) around the value. There are no limitation in the size of the text. The only limitation is the 16mb for the whole document. The larger the text the larger the data it takes.

Notice how numbers and booleans do not require a quotation marks around the value.

There are different types of numbers in mongoDB. Integer (int32) are 32bit long numbers and if we try to store a number longer than this they would overflow that range and we will end up with a

different number. For longer integer numbers we would use NumberLong (int64). The integer solution we decide to choose will dictate how much space will be allocated and eaten up by the data. Finally, we can also store NumberDecimal i.e. numbers with decimal values (a.k.a float in other programming languages).

The default within the shell is to store a int64 floating point value but we also have a special type of NumberDecimal provided by mongoDB to store high precision floating point values. Normal floating point values (a.k.a doubles) are rounded and are not precise after their decimal place. However, for many use cases the floating point (double) is enough prevision required e.g. shop store. If we are performing scientific calculations or something that requires a high precision calculation, we are able to use the special type that offers this very high decimal place precision (i.e. 34 digits after the decimal place).

The ObjectId is a special value that is automatically generated  by mongoDB to provide a unique id but it also provides some temporal component that allows for sorting built into the ObjectId, respecting a timestamp.

The above table provides all the data types within mongoDB that we can use to store data within our database server.

# Data Types & Limits

MongoDB has a couple of hard limits. The most important limitation: a single document in a collection (including all embedded documents it might have) must be less than or equal to 16mb. Additionally we may only have 100 levels of embedded documents.

We can read more on all the limitation (in great detail) on the below link:

https://docs.mongodb.com/manual/reference/limits/

For all the data types that mongoDB supports, we can find a detailed overview on the following link:

https://docs.mongodb.com/manual/reference/bson-types/

**Important data type limits are:**

- Normal Integers (int32) can hold a maximum value of +-2,147,483,674
- Long Integers (int64) can hold a maximum value of +-9,223,372,036,854,775,807
- Text can be as long as we want – the limit is the 16mb restriction for the overall document.

It's also important to understand the difference between int32 (NumberInt), int64 (NumberLong) and a normal number as you can enter it in the shell. The same goes for a normal double and NumberDecimal.

NumberInt creates a int32 value => NumberInt(55)

NumberLong creates a in64 value => NumberLong(7489729384792)

If we just use a number within the shell for example insertOne( {a: 1} ), this will get added as a normal double into the database. The reason for this is because the shell is based on JavaScript which only knows float/double values and does not differ between integers and floats.

NumberDecimal creates a high precision double value => NumberDecimal("12.99")

This can be helpful for cases where we need (many) exact decimal places for calculations.

When working with mongoDB drivers for our application's programming language (e.g. PHP, .NET, Node.js, Python, etc.), we can use the driver to create these specific numbers. We should always browse the API documents for the driver we are using within our applications to identify the methods for building int32, int64 etc.

Finally we can use the db.stats() command in the mongoDB shell to see stats of our database.

# How to Derive Our Data Structure Requirements

Below are some guidelines to keep to mind when we think about how to structure our data:

- What data does our App need to generate? What is the business model?

User Information, Products Information, Orders etc. This will help define the fields we would need (and how they relate).

- Where do I need my data?

For example, if building a website do we need the data on the welcome page, products list page, orders page etc. This help define our required collections and field groupings.

- Which kind of data or information do we want to display?

For example the welcome page displays product names. This will help define which queries we need i.e. do we need a list of products or a single product.

These queries we plan to have also have an impact on our collections and document structure. MongoDB embraces the idea of planning our data structure based on the way we retrieve the data so that we do not have to perform complex joins but we retrieve the data in the format or almost in

the format we need it in our application.

- How often do we fetch the data?

Do we fetch data on every page reload, every second or not that often? This will help define whether we should optimise for easy fetching of data.

- How often do we write or change the data?

Do we change or write data often or rarely will help define whether we should optimise for easy writing of data.

The above are things to keep in mind or to think about when structuring our data structures and schemas.

### Schemas & Relations: How to Structure Documents

## Understanding Relations

Typically we would have multiple collections for example a users collection, a product collection and a orders collections. If we have multiple collections that are relatable or where the documents in these relations are related, we obviously have to think about how do we store related data.

Do we use embedded documents because this is one way of reflecting a relation or alternatively, do we use references within our documents?

| Nested/ Embedded Documents | References |
| --- | --- |
| **Customers Collection** | **Customers Collection:** |
| { | { |
| “userName”: “John”, | “userName”: “Alan” |
| “age”: 28, | “favBooks”: [“id1”, “id2”] |
| “address”: { | } |
| “street”: “First Street”, | **Books Collection:** |
| “City”: “Chicago” | { |
| } | “_id”: “id1”, |
| } | “name”: “Lord of the Rings” |
| | } |

In the reference example above, we would have to run two queries to join the data from the different collections. However, if a book was to change, we would only update it in the books collection as the id would remain the same whereas in a embedded document relation we would have to update multiple customer records affected with the new change.

# One to One Embedded Relation Example

**Example:**

One patient has one disease summary, a disease summary belongs to one patient.

**Code snippet:**

```
$ use hospital
$ db.patients.insertOne( { name: "John Doe", age: "25", diseaseSummary: { diseases: ["cold", "sickness"] } } )
```

Where there is a strong one to one relation between two data, it is ideal to use a one to one embedded approach as demonstrated in the above example.

The advantage of the embedded nested approach is that within our application we only require a single find query to fetch the necessary data for the patient and disease data from our database collection.

# One to One Reference Relation Example

**Example:**

One person has one car, a car belongs to one person.

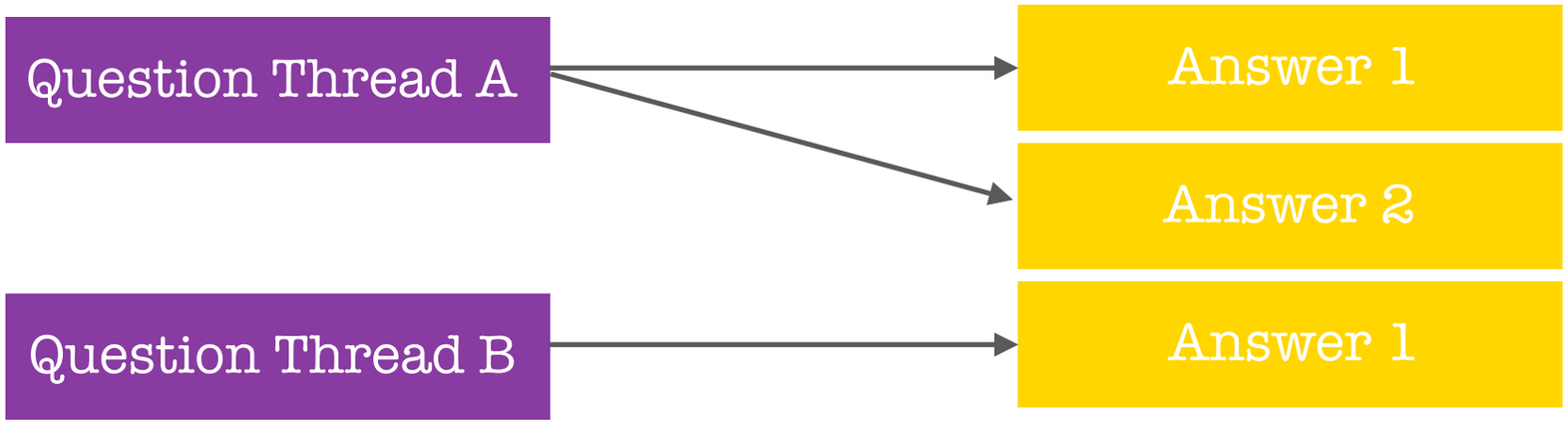
**Code snippet:**

```
$ use.carData
$ db.persons.insertOne( { name: "John", age: 30, salary: 30000 } )
$ db.cars.insertOne( { model: "BMW", price: 25000, owner: ObjectId("5b98d4654d01c") } )
```

In most one to one relationships we would generally use the embedded document relations. However, we can opt to use a reference relation approach as we are not forced to use one approach.

For example, we have a more analytics use case rather than a web application and we have a use case where we are interested in analysing the person data and or analysing our car data but not so much in a relation. In this example we have a application driven reason for splitting the data.

# One to Many Embedded Relation Example




**Example:**

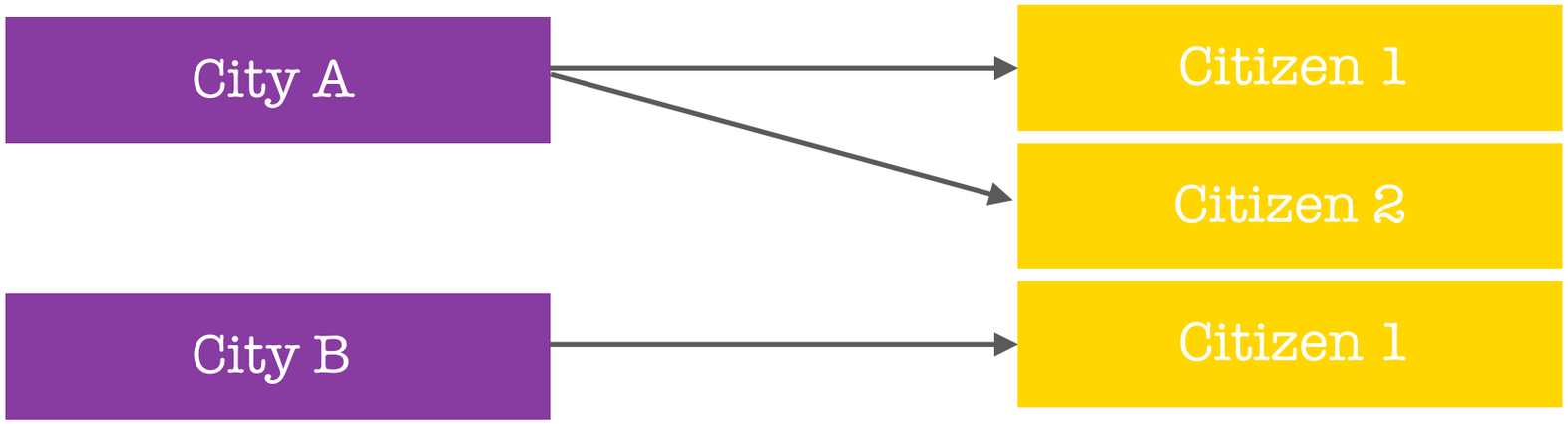
One question thread has many answers, one answer belongs to one question thread.

**Code snippet:**

```
$ use support
$ db.questionThreads.insertOne( { creator: "John", question: "How old are you?", answers: [ { text: "I am 30." }, { text: "Same here." } ] } )
```

A scenario where we may use a embedded one to many relation would be post and comments. This is because you would often need to fetch the question along with the answers in an application perspective. Also usually there are not too many answers to worry about the 16mb document limit.

# One to Many Reference Relation Example




**Example:**

One city has many citizens, one citizen belongs to one city.

**Code snippet:**

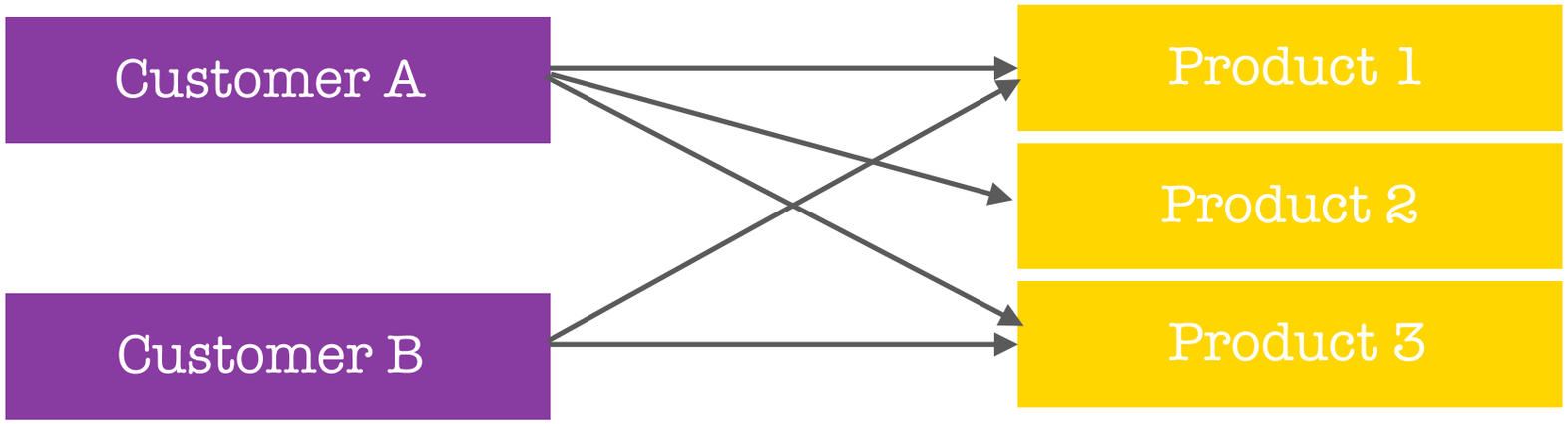
```
$ use cityData
$ db.cities.insertOne( { name: "New York City", coordinates: { lat: 2121, lng: 5233 } } )
$ db.citizens.insertMany( [ { name: "John Doe", cityId: ObjectId("5b98d6b44d") }, { name: "Bella Lorenz", cityId: cityId: ObjectId("5b98d6b44d") } ] )
```

In the above scenario we may have a database containing a collection of all major cities in the world and a list of every single person living within that city. It would seem to make sense to have a one to many embedded relationship, however, from an application prospective we may wish to only retrieve the city data only. Furthermore, a city like New York may have over 1 million people data and this would make fetching the data slow due to the volume of data passing through the wire. Furthermore, we may end up running into the document size limit of 16mb. In this type of scenario, it would make sense to split the data up and using the reference relation to link the data.

In the above we would only store the city metadata and will not store any citizen reference as this will also end up being a huge list of citizens unique id. Instead, we would create a citizens collection and within the citizens data we would make reference to the city reference. The reference can be anything but must be unique ie. we could use the ObjectId() or the city name etc.

This will ensure that we do not exceed the limitation of the 16mb per document as well as not retrieving unnecessary data if we are only interested in returning just the cities metadata from a collection.

# Many to Many Embedded Relation Example




**Example:**

One customer has many products (via orders), a product belongs to many customers.

**Code snippet:**

```
$ use shop
$ db.products.insertOne( { title: "A Book", price: 12.99 } )
$ db.customers.insertOne( { name: "Cathy", age: 18, orders: [ { title: "A Book", price: 12.99, quantity: 2 } ] } )
```

We would normally model many to many relationships using references. However, it is possible to

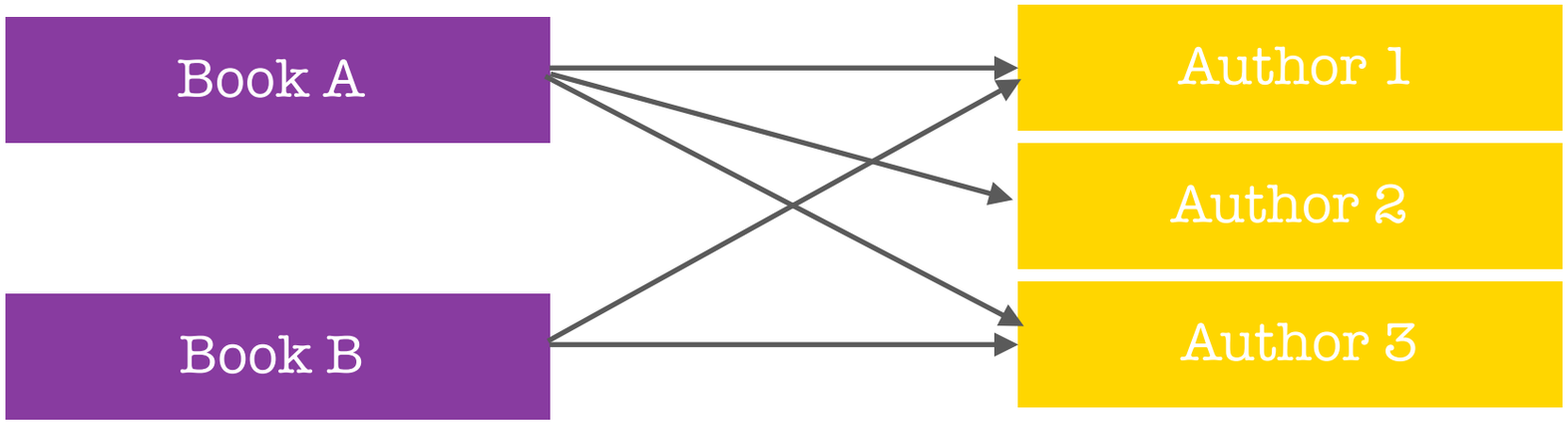
use the embedded approach as seen above. We could store a collection for the products as meta data for an application to retrieve the data in order to help populate the embedded document of the customer collection using a programming language.

A disadvantage to the embedded approach is data duplication because we have the title and price of the product within the orders array as the customer can order the product multiple times as well as other customers which will cause a lot of duplication.

If we decide to change the data for the product, not only do we need to change it within the product collection but we also have to change it on all the orders affected by this change (or do we actually need to change old orders?). If we do not care about the product title changing and the price changing i.e. we have an application that takes a snapshot of the data, we may not worry too much about duplicating that data because we might not need to change it in all the places where we have the duplicated the data if the original data changes – this highly depends on the application we build. Therefore a embedded approach may work.

In other case scenarios where we absolutely need the latest data everywhere, a reference approach may be most appropriate in a many to many relationship. It is important to think about how we would fetch our data and how often do we want to change it and if we need to change it everywhere or are duplicate data fine before deciding which approach to adopt for many to many.

# Many to Many Reference Relation Example




**Example:**

One book has many authors, an author belongs to many books.

**Code snippet:**

```
$ use bookRegistry
$ db.books.insert( { name: “favourite book”, authors: [ objectId(“5b98d9e4”), objectId(“5b98d9a7”) ] } )
$ db.authors.inserMany( [ { name: “Martin”, age: 42 }, { name: “Robert”, age: 56 } ] )
```

The above is an example of a many to many relation where a reference approach may be suitable for a scenario where the data that changes needs to be reflected everywhere else.

# Summarising Relations

We have now explored the different relation options that are available to use. This should provide us enough knowledge to think about relations and when to use the most appropriate approach depending on:

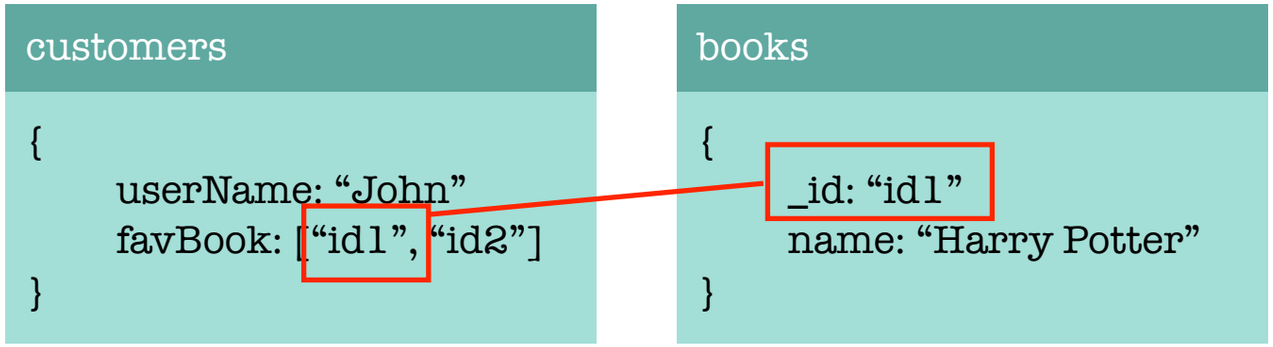
- the application needs
- how often data changes
- if a snapshot data suffice
- how large is the data (how much data do we have).

**Nested/Embedded Documents** – group data together logically. This makes it easier when fetching the data. This approach is great for data that belong together and is not overlapping with other data. We should always avoid super-deep nesting (100+ levels) or extremely long arrays (16mb size limit per document).

**References** – split data across collections. This approach is great for related data but also shared data as well as for data which is used in relations and standalone. This allows us to overcome nesting and size limits (by creating new documents).

# Using $lookup for Merging Reference Relations

MongoDB has a useful operation called $lookup that allows us to merge related documents that are split up using the reference approach.




The image on the right provides a scenario of a reference approach where the customer and books have been split into two collections. The lookup operator is used as seen below. This uses the aggregate method which we have not currently learned.

```
$ customer.aggregate( [
    { $lookup: { from: “books”, localField: “favBooks”, foreignField: “_id”, as: “favBookData” } }
] )
```

The $lookup operator allows us to fetch two related documents merged together in one document within one step (rather than having to perform two steps). This mitigates some of the disadvantages of splitting our documents across multiple collections because we can merge them in one go.

This uses the aggregate method framework (which we will dive into in later chapters) and within the aggregate we pass in an array because we can define multiple steps on how to aggregate the data. For now we are only interested in one step (a step is a document we pass into an array) where we pass the $lookup step. The lookup passes in a document as a value, where we define 4 attributes:

- **from** – which other collection do we want to relate documents i.e. we would pass in the name of the collection where the other document lives that we wish to merge.
- **localField** – in the collection we are running the aggregate function on, where can the reference to the other (from) collection be found in i.e. the key that stores the reference.
- **foreignField** – which field are we relating to in our target collection (i.e. the from collection)
- **as** – provide an alias for the merged data. This will become the new key which the merged data will sit.

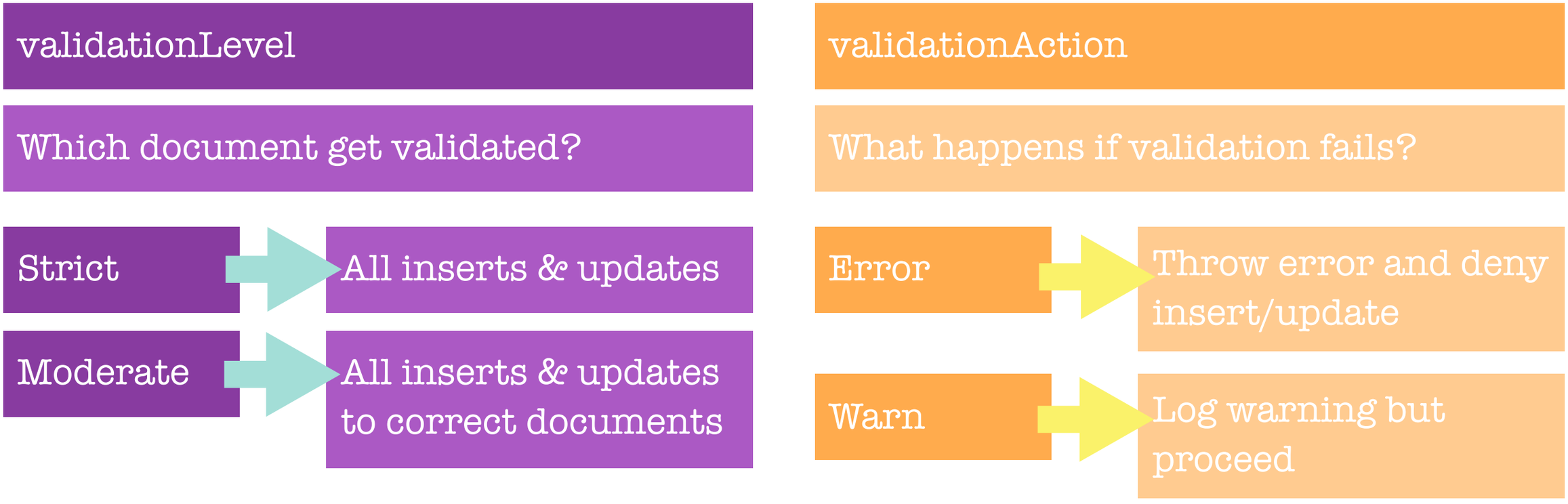
This is not an excuse to always using a reference relation approach because this costs more performance than having an embedded document.

If we have a references or want to use a references, we have the lookup step in the aggregate method that we can use to help get the data we need. This is a first look at aggregate and we will explore what else the aggregate can do for us in later chapters.

# Understanding Schema Validation

MongoDB is very flexible i.e. we can have totally different schemas and documents in one and the same collection and that flexibility is a huge benefit. However, there are times where we would want to lock down this flexibility and require a strict schema.

Schema validation allows mongoDB to validate the incoming data based on the schema that we have defined and will either accept the incoming data for the write or update to the database or it will reject the incoming data and the database is not changed by the new data and the user gets an error.



# Adding Collection Document Validation

To add schema validation in mongoDB and the easiest method is to add validation when we create a new collection for the very first time explicitly (not implicitly when we add a new data). We can use the createCollection to create and configure a new collection:

```
$ db.createCollection("posts", { validator: { $jsonSchema: { bsonType: "object", required: ["title", "text", "creator", "comments"], properties: { title: { bsonType: "string", description: "must be a string and is required." }, text: { bsonType: "string", description: "must be a string and is required" }, creator: { bsonType: "objectId", description: must be an objectId and is required }, comments: { bsonType: "array", desription: "must be an array and is required", items: { bsonType: "object", required: ["text,"], properties: { text: { bsonType: "string", description: "must be a string and is required" }, author: { bsonType: "objectId", description: "must be an objectId and is required" } } } } } } } )
```

The first argument to the createCollection method is the name of the collection i.e. we are defining the name of the collection. The second argument is a document where we would configure the new collection. The validator is an important piece of the configuration.

The validator key takes in another sub document where we can now define a schema against incoming data where inserts and updates has to validated. We do this by inserting a $jsonSchema key with another nested sub document which will hold the schema.

We can add a son type with the value of object, so that everything that gets added to the collection should be a valid document or object. We can set a required key which has an array value. In this array we can define names of fields in the document which will be part of the collection that are absolutely required and if we try to add data that does not have these fields, we will get an error or warning depending on our settings.

We can add a properties key which is another nested document where we can define how for every property of every document that gets added to the collection will look like. In the example above we defined the title property, which is a required property, in more detail. We can set the bsonType which is the data type i.e. string, number, boolean, object, array etc. We can also set a description for the data property.

Because an array and has multiple items, we can add an items key and describe how the items should look like. We can nest this and this can have another nested required and properties keys for the items objects that exists within the array.

So the Keys to remember are:

The bsonType key is the data type.

The required key is an array of required properties that must be within an insert/update document.

The properties key defines the properties. This has sub key:value of of bsonType and description.

The Item key defines the array items. This can have sub key:value of all the above.

**Important Note:** it may be difficult to read in the terminal and may be easier to write in a text editor first and then paste into the terminal to execute the command. We can call the file validation.js to save the collection validation configuration. Visual Studio/Atom/Sublime or any other text editor/ IDE will help with auto-formatting. Visual Studio has a option under code > Preference > Keyboard Shortcuts and then you can search for a shortcut command such as format document (shortcut is Shift + Option + F on a Mac).

We can now validate the incoming data when we explicitly create the new collection. We can copy the command from the text editor and paste it back into the shell and run the command to create the new collection with all our validation setup. This will return { "OK" : 1 } in the shell if the new collection is successfully created.

If a new insert/update document fails the validation rules, the new document will not be added to the collection.