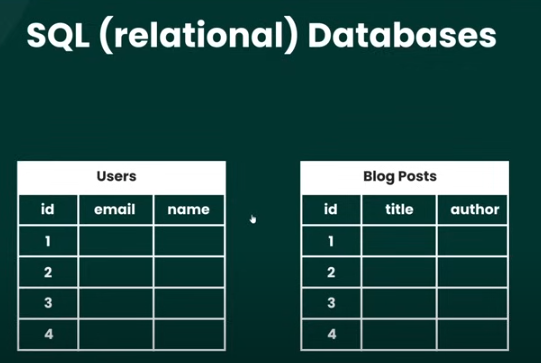
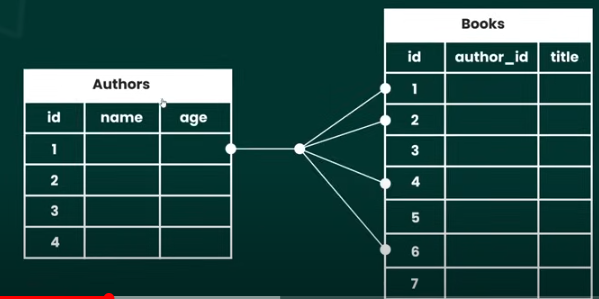
**1. Introduction**

* Is a NoSQL database, meaning we don’t use “SQL” command to interact with it.
* SQL-databases(Relational databases) such as MySQL, are made up of tables with rows and columns, where each tables is used to store specific data type like users or blog post.
  + each row is a record, eg. User record,
  + Each column is a property on that particular record (like username, email, name,gender, …)



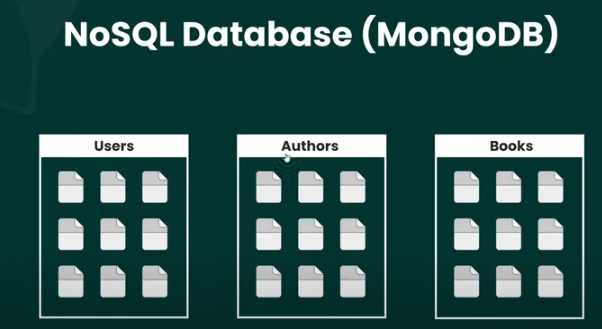
* + Often times these datatypes in data tables are related to each other(forming a relationship between tables). For example, author in one table may have authored multiple books, so we will have one-to-many relationships between ‘Author” and “Books” table.



* + To query this SQL databases, we use SQL commands



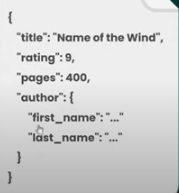
* Unlike SQL-databases (Mysql…) which uses Instead of Tables, rows and Columns to store data, MongoDB(NoSQL) database uses Collections and documents to store data.
  + Each collection in NoSQL database store specific type of data record like users, authors, books, blog posts….



* + Records inside collections are called documents. They look like JSON objects(key-value pairs)



* + Benefits of using structure like Json over tables, columns and row
    - Easier to work with JavaScript, because JSON like structure of data storing
    - Allow us to store nested documents within document. Alternative to the approach of using two different tables in SQL database to store those data types separately.

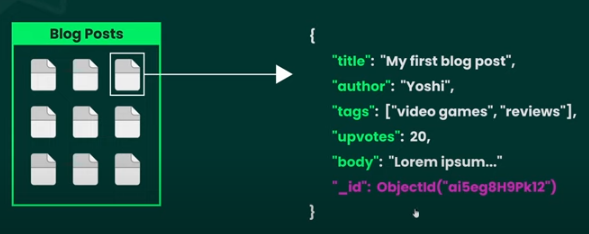


**2- Installation and Setup**

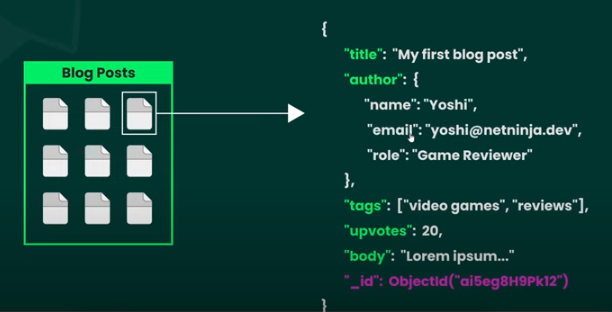
* Installing MongoDB Locally
  + Go to <https://www.mongodb.com/try/download/community>
  + Download the current version of community database for your platform(in this case windows) and the package type “msi” (not Zip ).
  + Run the installer > Next > Accept the Terms > Complete > Check Install MondoDB as a service, and Run Service as Network service user radio (take default Values) > check Install MongoDB compass( is GUI Interface to work with DB) and click Next > Install > finish.
  + When done MongoDB Compass might auto start for you, before we use it we need to install the shell first
* Install MongoDB Shell
  + Allows as to interact with MongoDB from a Terminal
  + Go to Tools option on the left Navigation where you have downloaded the community version DB earlier, here you can find tools to work with MongoDB including the Compass(since we have installed it during DB installation we don’t install it here again)
  + Select the MongoDB Shell Option, download latest version for your platform(Windows x64) with ‘msi”(not Zip) Package type.
  + Start the installer by double clicking > Next > Next > Install > Finish
  + Verify the installation: Open Cmd >#mongosh //If installed correctly this will start interactive mongo shell stating versions and other details then with new prompt line

**3. Collections and Documents**

* MongoDB stores data inside collections, and a database can have as many collections (Groups) as you like for different types of data.
* Documents represent individual record in specific collection.
  + For example, inside blog post collection we would store load of blog post documents and each one represent single blog post.

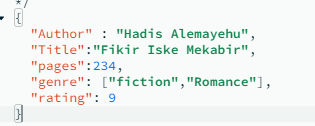


* + Ever document uniquid Id Propery to identify them, which is special Object Id type assigned to the document by the MongoDB itself when we create a document.
  + Documents are represented by bytes (BSON) which looks like JSON objects.
  + Documents can have properties whose values themselves are documents or Arrays of documents=> *Nested documents*



**4-Using MongoDB Compass**

* GUI interface to Visualize and manipulate Mongo databases.
* Show us all of our databases we created and all the Collections in those databases and all documents inside collections as well.
* Allow as create collections, add new documents, update existing documents or delete them.
* Basic operations:
  + Creating Connection for first use:
    - Open MongoDB Compass > Add new Connections
    - Connection string is special URL used to connect to MongoDB Cluster
    - If you use MongoDB service such as “Atlas”(allows you to create cluster online and have your database hosted online), you paste the connection string for that cluster obtained from there, here in this input field. Then MongoDb Compass will connect to it and you can use this GUI tool to interact with the databases served on atlas.
    - However, If you want to connect to MongoDB installed locally just click Connect Button, With the default connection string(*mongodb://localhost:27017*) set for you. This will automatically connect you to local MongoDB service and list all available mongo databases you have on the machine along the premade databases by the mongo( such as admin, config, local).
      * For example local DB contains database startup data, when we started the service on the computer
    - If you run into error, Start the mongo service running on your computer.
      * Search > Services > Right Click and start MongoDB Server if it the status says stopped.
  + Creating Database:
    - Under connections click localhost > Create database button
    - Provide the database name that identifies the DB, for example if you are making database for bookstore website give a name “bookstore”.
    - When Creating new DB, we need to add at least one collection. Lets provide “book”
    - Inside database we can have multiple collections for different types of data.
    - Then click the create Database button, this will create the database add it to the already existing databases on the left of the compass.
    - Clicking the database name will open DB overview window that display the content including the collection contained in that DB.
      * If you want to add new collection to the DB, Click “create collection” button an d provide the collection name to the input textbox.
      * Or Click the plus(+) sign Infront of target database name and give collection name on the next window
  + Add Document/Data to the collection
    - Click on the collection name (ex. Book ) to open the collection > from the drop down select insert document option if you want to add data manually or choose “import JSV or “csv” if you already have such file containing the data
    - choosing “insert document”: will open new window having JSON like structure with unique ID generated for the document. (if you remove this ID, it will automatically inserted when you click “insert” button.
    - So start inserting data to the object with key-value pair and click insert button

D

* + - Note, the \_id is automatically inserted by MongoDB
    - If you want to edit or delete each document, hover of the document entry and the icons for editing will show up, choose the one correspond to your action.
    - To filter the data, specify the criteria on top textbox field provided below toolbar
      * Example: {rating:9} then click find //returns documents that have rating=9
  + Deleting DB
    - Click the delete(recycle bin ) icon Infront of the database name and confirm on popup.

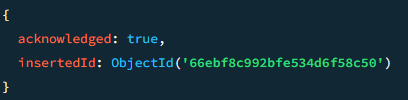
**5-Using MongoDB Shell**

* Used to interact with the DB via terminal, to create, fetch, update and delete data
* This approach is closer to the way we interact from application code with DB
* The shell can be accessed from ***bundled tool in to the compass tool*** or from any other terminal on your computer(after installing mongoShell on your computer).
* To run mongo shell on your computer terminal, you need mongo shell installed on your computer(refer chapter two: installation section)
* Common mongosh commands:
  + *#test > show dbs* //lists all the available mongo databases on our server
    - The “test” shows the current database we are working with//is default DB
  + # *use bookstore* //Switches to the DB named bookstore
    - notice the cursor will change from “ test >” to “bookstore >” specifying the target DB
  + If you want to leave the current DB : # use test
  + Note, The “test” DB doesn’t exist. In the shell It doesn’t matter if DB exists or not, before we start work with it.
    - For example, insert #use mydb in the shell it switches to it, even though the DB doesn’t exist. technically you can switch between DBs even though the DB doesn’t exist.
  + To Create the DB you should add Collection and add data/documents to the collection, when you do that the DB will be created for us.
  + Using another Terminal such as windows ***CMD***(command prompt)to Run mongosh commands
    - Open the Terminal (command Prompt) > # mongosh //opens the mongo shell
    - #*cls* //clears the screen
    - # Bookstore > *db* // show the current DB we are working with
    - # Bookstore > *show collections*//lists all the collections inside current DB
    - # Bookstore > *var name=” Jamal”* // creates variable and store value to it
    - # Bookstore > *name* // returns the value of the variable “name”
    - # Bookstore > *var name=” Doe”* // reassign the value of “name”
    - # Bookstore > *help* // help on the commands
    - #Bookstore > *exit* // exit the shell
  + Add Documents to DB
    - test> *use Bookstore* // switch to the target DB(which is “Bookstore”)
    - Bookstore > *db.books* // targets the collection in current Db. ( note #db returns current database we are working with)



//the *db.books.insertOne( )* method creates single record data(document) in the “books” collection, with object specified in the parenthesis. In addition unique Id for this document will be automatically generated and added along the document .

* + - Then the hell confirm the insertion of the document by generating something like this:



* + - Note, the collection doesn’t need to exist to insert data to it. For example the following if you run the following command on the “Bookstore” database that does not have “authors” collection, the command will execute without a propbem, even through the “authors” collection doesn’t exist”
      * db.authors.insertOne({name:"Addis Alemayehu", age:76})
      * However, when you execute the command the collection will be created automatically in the target database. Verify by #show collections
    - You can also insert multiple documents at once using db.books.insertMany( ) method, which takes an array of objects(documents)
      * #Bookstore >*db.books.insertMany([ {title:"Bonga",author:"Konta Tigebaleh", pages:50, ratings:4,genres:["commedy","Fun"]},{title:"Gadda System", author:"Dr. Gamachu",pages:700,ratings:10,genres:["History","Seena"]}])*

//confirmation message with auto generated Unique ID for both documents we just inserted

{

acknowledged: true,

insertedIds: {

**'0'**: ObjectId(**'66ebfff892bfe534d6f58c51'**),

**'1'**: ObjectId(**'66ebfff892bfe534d6f58c52'**)

}

}

**7.Finding Documents in the Collection**

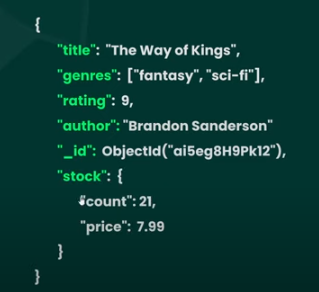
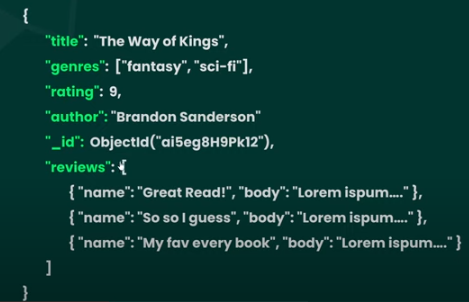
* # Bookstore>db.books.find( )// will return the first **20** documents in the “books” collection
* # Bookstore >db.books.find({author:"Dr. Gamachu"}) // filters and returns all the books authored by “Dr. Gamachu”
* ***Note, filters are arguments passed to find() to specify what kind of values we want to retrieve.***
* # *Bookstore> db.books.find({author:"Dr. Gamachu",ratings:10})* // we can specify multiple chriteria, for example in this case it filters books written by “Dr. Gamachu” and have ratings of 10
* # *Bookstore>* db.books.find({author:"Dr. Gamachu"},{title:1,author:1}) // the second agreement in this find method specifies the fields of the document we want( in this case title and author’s name)
  + Note the id is returned even if we don’t query it
* If you want to retrieve only the specific fields of all the documents, pass empty bracket for the first argument:
  + # *Bookstore>db.books.find({ },{title:1,author:1})//*No filter, just selected 2 fields for all documents
* If you want to find specific document use findOne( ) method by passing the “ID” as argument
  + #*Bookstore> db.books.findOne({ \_id: ObjectId('66ebfff892bfe534d6f58c51')})*

**8-Sorting and Limiting Data**

* We can tag on extra method(method chaining) to the find( ) method to Perform additional tasks like sorting data, Limiting the amount of documents to retrieve.
* *# Bookstore> db.books.find().count()* //Counts and returns the total number of documents in “books” collection.
* # *Bookstore> db.books.find({Author:"Hadis Alemayehu"}).count()* // returns the number of books authored by “Hadis Alemayehu”
* # Bookstore> db.books.find().limit(3) //limits the query to only return 3 documents at a time
* Sorting
  + Bookstore> db.books.find().sort({title:1}) //sorts documents in asending order or the “title”, if you want discending use “-1” in place of 1
  + # *Bookstore> db.books.find().sort({title:-1}).limit(3)* //sort in descending order and returns only three documents
  + # Bookstore> db.books.find().sort({title:-1}).limit(3).count() //counts the returned documents, obviously we get 3, since it is specified in limit() method.

**9-Nested Documents**

* A document is JSON like Object having Key-value pair, and the Value can be any kind of data such as integer, string, array or objects(or nested document).

or 

* Improve read performance

**10-Operators and Complex Queries**

* Recall the find( ) methods and filters passed as arguments to the find() methods, looks for exact match of the value when querying the documents. For example *db.books.find({ratings:7})* will find all the book with the ratings exactly “7”.
* But what if we want to look for books with the rating of 7 and above/less. We will use query operators for such cases.
* Operators in MongoDB are denoted by dollar($) sign(i.e. all operators begin with “$” sign).

Examples

* + # *Bookstore> db.books.find({ratings:{$gt:7}})* //returns all the books with ratings greater than seven but **not** including “7”
  + *Bookstore> db.books.find({ratings:{$gte:7}})* // all values >=7 (including 7)
  + *Bookstore> db.books.find({ratings:{$lt:8}})* // returns all the books with ratings less than eight but **not** including “8”
  + *Bookstore> db.books.find({ratings:{$lte:8}})* //all values <=8 (including 8)
* Multiple different queries/filters are also allowed to further refine the search
  + *Bookstore> db.books.find({ratings:{$gte:7},author:"Mel Gibson"})* // all books that have ratings >=7 and written by “Mel Gibson”
* Using “**or**” operator to query matching either of the filters specified
  + *db.books.find({$or:[{ratings:7},{ratings:9}]})* // returns books that have ratings of 7 or 9
  + Bookstore> db.books.find({$or:[{ratings:7},{author: "John Doe"}]}) // returns books which have rating of “7” and books that is authored by “John Doe”
  + *Bookstore> db.books.find({$or:[{pages:{$lt:200}},{pages:{$gt:400}}]})* // returns books that have pages less than 200 and those are greater than 400(i.e. both )

**11.Using $in and $nin Operators**

* To filter fields that are in/not in particular **range** of values (or array of values)
* ***Using “$in” operator***
  + Bookstore> db.books.find({ratings:{$in:[6,7,8]}}) // get all books that have ratings 6, 7 and 8
  + The above query is equivalent to the Following query with “$or” operator
  + # Bookstore> db.books.find({$or:[{ratings:6},{ratings:7},{ratings:8}]})
* ***Using “$nin”(not in) operator*** : is opposite to the “$in” operator
  + # *Bookstore> db.books.find({ratings:{$nin:[6,7,8]}})* // returns all the books that do not have the rating of 6,7 or 8 (i.e. its result is opposite to the $in operator)

**12. Querying Arrays**

* Check if the specified value exists in the array
  + # Bookstore> db.books.find({genres:"fantsy"}) //returns books containing genres of “fantasy”, note this option is used like wild card, because it doesn’t look for exact match(the **book might have other genres beside fantasy) . Means, it doesn’t filter books that only has** “fantasy”
* Checks if the array contains only the specified value(exact match)
  + # Bookstore> db.books.find({genres:["fantacy"]}) // put bracket around fantasy to get the exact match(i.e. books that only have fantasy as genres)
  + You Can also query for multiple values matching the filter :
  + Bookstore> db.books.find({genres:["fantacy","action"]}) // books with genres of fantasy and action
* Checking document if an array have both the values at same time (Use “$all” operator)
  + # Bookstore> db.books.find({genres:{$all:["fantacy","action"]}}) // the book returned must have both “fantasy” and “Action” as genres
* Querying array of objects (i.e. objects in array), we will use dot(.) notation to access nested values
  + # *Bookstore> db.books.find({"reviews.name": "James Cameron"})*
  + Note, we the attribute name(key) should be inside quotation when using dot notation

13. Deleting Documents

* Exporting collection via compass:
  + Export Data > Export Full collection >choose the file format (either JSON or CSV) >Export(btn) > choose Location and save the exported file
* Import Data via Compass:
  + Add Data>Import JSON of CSV file >browse the Location of the file and confirm
* Deleting One Document using deleteOne( ) Method
  + Note we use the unique ID generated by mongo for the document on the creation. You can use other keys(attribute) too, but since those fields are not unique you might end up deleting more than one document. So “\_id” is preferable

# *Bookstore> db.books.deleteOne({\_id:ObjectId('66ec46d07709b1cc46c73bfa')})*

* + if succeeded you will get confirmation like the following

{ acknowledged: true, deletedCount: 1 }

* Deleting Many Documents using “***deleteMany( )”*** Method
  + *Bookstore> db.books.deleteMany({author:”John Doe”)})* // deletes all the books authored by John Doe
  + The confirmation message will let us know how many documents are deleted

{ acknowledged: true, deletedCount: 4 }

13. Deleting Documents

* Updating One Document using updateOne( ) Method
  + Note we use the unique ID generated by mongo for the document on the creation.

# *Bookstore> db.books.updateOne({\_id: ObjectId('66ec68787709b1cc46c73bfd')}, {$set:{ratings:9,pages:10}})*

Note the updateOne( ) method takes two objects as argument, the first to specify/target the document we want to modify and the second uses the set ($set) operator to set as many different fields with corresponding value we want.

* + if succeeded you will get confirmation like the following telling us the count of document modified along other inforamtion

{

acknowledged: true,

insertedId: null,

matchedCount: 1,

modifiedCount: 1,

upsertedCount: 0

}

* Update Many Documents using “***updateMany( )”*** Method, to modify multiple document at once
  + *Bookstore> db.books.updateMany({author:"Mel Gibson"},{$set:{author:"Mel Gibirsonser"}})*

// modifies all the books with author name “Mel Gison” to “Mel Gibirsonser”

* + The confirmation message will let us know how many documents are updated in this case 2

{

acknowledged: true,

insertedId: null,

matchedCount: 2,

modifiedCount: 2,

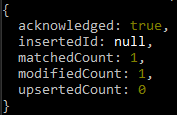
upsertedCount: 0

}

* **Using “$inc” Operator to increament a value**

# Bookstore> db.books.updateOne({ \_id: ObjectId('66ec68787709b1cc46c73bfd')},{$inc:{pages:12}})

* + Increments the value of pages by 12 for the document specified by ID in the first argument
  + the confirmation message tell count of document affected like the folloing



* ***you can also use the “$inc” Operator to decrement(reduce) by providing Negative value***

# Bookstore> db.books.updateOne({ \_id: ObjectId('66ec68787709b1cc46c73bfd')},{$inc:{pages:-1}})

* + Notice we give negative value to decrement a pages field
* **Removing Values from an array via “$Pull” Operator**

# Bookstore> db.books.updateOne({\_id: ObjectId('66ec68787709b1cc46c73bfd')}, {$pull:{genres:"revenge"}})

//removes(pulls) the “revenge” genres from a book specified by its ID in the first argument

* **Add Values to an array via “$Push” Operator**

# Bookstore> db.books.updateOne({\_id: ObjectId('66ec68787709b1cc46c73bfd')}, {$push:{genres:"fantacy"}})

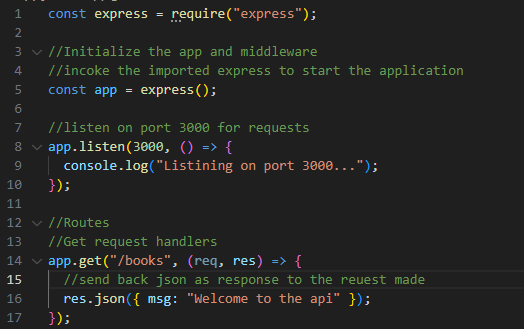
//adds(pushes) the “fantacy” genres to the book specified by its ID in the first argument

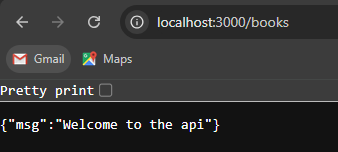
* **Push or pull multiple different items to/from an array via “$each” Operator along the “$push” or “$pull” operator**

Bookstore> db.books.updateOne({\_id: ObjectId('66ec68787709b1cc46c73bfd')}, {$push:{genres:{$each:["genr1","gebr2"]}}})

//adds two values(genr1 and gebr2) to genres field of specified book in first argument

**15.MongoDB Drivers**

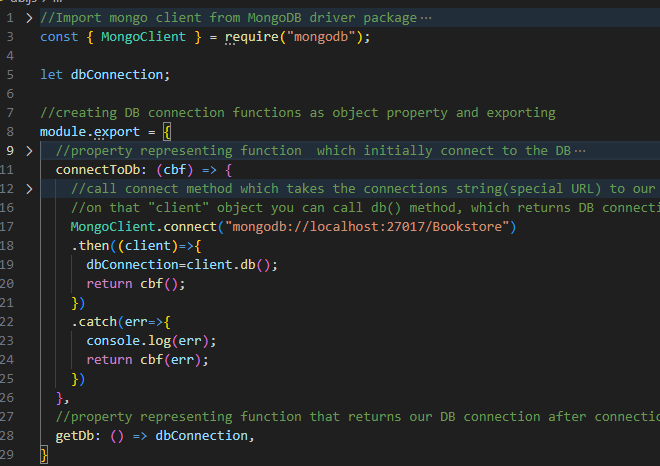
* Provides specific languages bindings when interacting with MongoDB form your applications. For example, if you are making python application you need python driver and if you are making NodeJS driver we need NodeJS driver to work with MongoDB.
  + Allows us to communicate with MongoDB on the programmatic level, to bridge the gab between your app and the mongoDB
* Obtaining these drivers and installing them
  + Go to <https://www.mongodb.com/> > Resources >MongoDB Drivers
  + Choose the programming language you used to develop your application (ex. NodeJS) then you will find the instruction how install and use it .
* learn NodeJS: <https://www.youtube.com/playlist?list=PL4cUxeGkcC9gcy9lrvMJ75z9maRw4byYp>
* For this exercise we will be using NodeJS
  + Install node on your machine first
  + Set up our application and install MongoDB driver
    - Open text editor such as VS code and create folder for your Project
    - Inside your directory(D:\TOP\MongoDB\mongo-app>) open terminal and initialize node project which creates “package.json” file which keeps track of our dependencies
      * # npm init
    - Take default values for the prompted question by pressing enter
    - Create entry file called “app.js” for our project in the project folder
    - Create Express app in this “app.js” file by installing express saving to dependencies
      * # npm install express –save
      * Import the express, set app with express and specify the port at which it listens to for requests,
      * define routes for your app as well .
    - 
    - Run the app
      * Install “nodemon” globally on your computer. It speed up development by watching changes made to the app code on save and restarts server automatically and let as notice the effect immediately.
        + # *npm install -g nodemon*
      * To run the app :
        + # *nodemon app*
      * Go to browser and open the following address,
        + <http://localhost:3000/books>



* + - Install MongoDB driver for node
      * Open terminal go to project folder and run the following command
        + # D:\TOP\MongoDB\mongo-app> *npm install mongodb --save*

**16. Connecting to MongoDB**

* Create new file (db.js) in your project folder
  + We need two functions in this file, one to initially connect to the database and one to retrieve that database connection once we have already connected to it, which allow as to interact with DB(add, remove, update data). Then export both functions from this file, so that we can use them in our “app.js” file.
  + <https://www.youtube.com/watch?v=gGNquGHqpNI&list=PL4cUxeGkcC9h77dJ-QJlwGlZlTd4ecZOA&index=16>

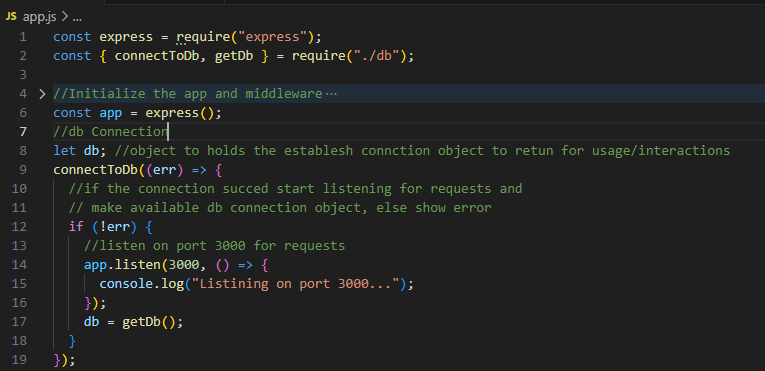


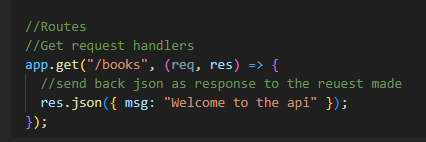
The code and full comment can be viewed on n your computer

* Call the two function created above to establish connection and return the established connection from “app.js”
  + Import the functions which are exported from “db.js” file

const {connectToDb, getDb}=require('./db');

* + Modify the “app.js” file to implement these imported files. On the implementation try connecting to the database and if the connection is successful start listening to requests, as well return the established connection as an object to interact with the DB for manipulations.





**17. Cursors and Fetching Data**

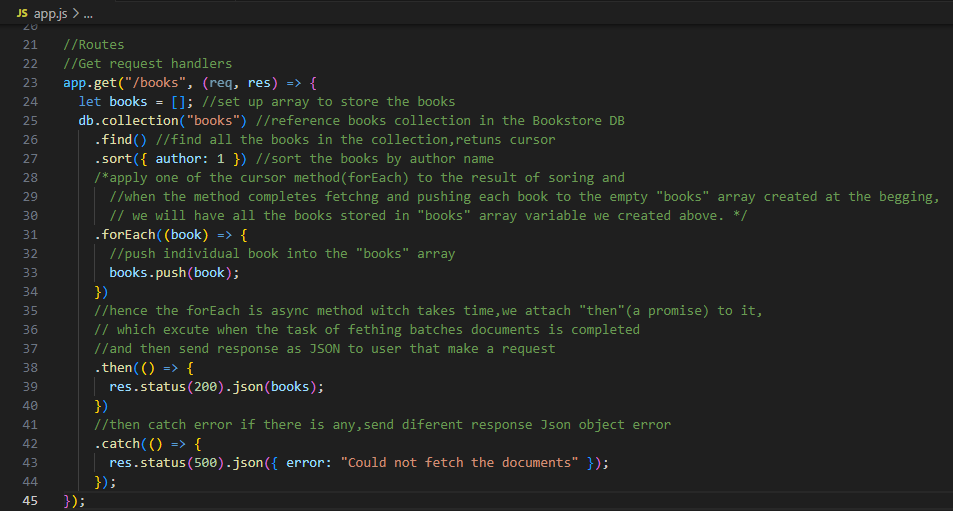
* Use the ‘db’ object that store the established connection in your app’s Get request handler to find data in the database collections and use them from your application.
  + Invoke the collection( ) method on “db” object by passing the target collection as string parameter to reference specific collection in the DB. this is equivalent to “*db.books”* command in the shell except we use collection() method which takes the collection name as string to target the collection we want to work with in NodeJS.

db.collection('books')

* + - Then call the find() method on it to find all the documents in the collection.

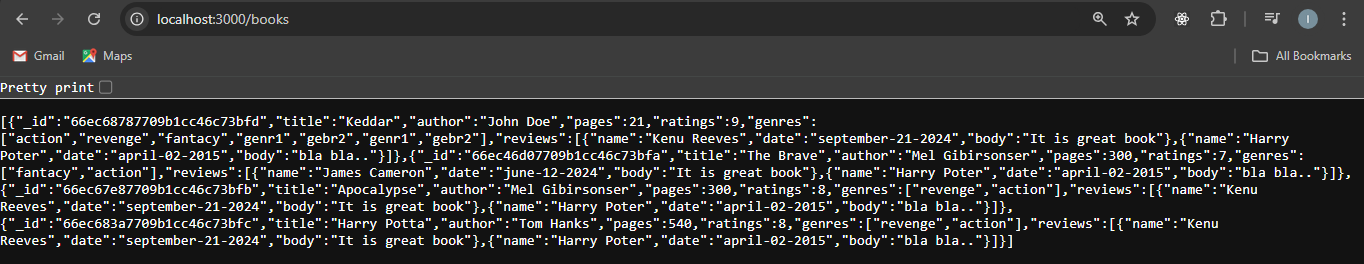
*db.collection('books').find()*

* + Find () Method and Cursor
    - ***Note***, when we used the find() method in the terminal, it seems like to automatically fetch all the documents and display them for us in the terminal. But actually find() method doesn’t return document for as, it returns a cursor.
    - **Cursor** is an object that point s to set of documents outlined by our query.
    - So without No argument inside the find() method, the cursor is going to point to the whole collection. But if we add filter as an argument, it will point to the subset of documents based on that filter.
    - This cursor object that is returned from the find() method, exposes methods that we can use to fetch the data which the cursor points to. Some of these methods we can sue are; toArray() and forEach() .
    - ***toArray( )*** fetches all of the documents the cursor points to and put them into an array.
    - ***forEach()*** iterates the document one at a time and then allows us to process each one individually
    - when we fetch documents from mongoDb using either of those methods, it actually gets documents from database in batches. That is because your collection could contain huge amounts of documents, if we fetch all those documents in one go it increase network bandwidth usage and affects performance. So MongoDB fetches the documents in smaller batches (the default is 101 of them at a time).
    - For example, if we use the *find()* method to get the cursor and on that cursor when we use “*forEach()”* method, it will fetch the first batch of documents and the “*forEach()”* method iterate batch of documents, so that we can process each one. Then after that batch is exhausted then fetch the nest batch so forth
    - Generally, when we use *find()* method it returns cursor that points to set of documents specified by filter or the entire collection if the filter is not applied. And we use method like *“forEach”* on that cursor to get the document and do something with it.
    - Note, we didn’t used this cursor methods when using *find()* in the shell, because the shell automatically iterated the first 20 documents for us and displayed them in the terminal. To see more use the “it” command on the terminal.
  + ***Sort()*** method sorts the documents based on the filters passed as argument
    - For example, lets sort the documents returned by the *find()* method by author name before applying the cursor method such as “forEach” on the resulting documents that iterates each book individually.



The complete code can be found on your computer or git repo(seena2)

* Try on the browser
  + Run # *nodemon app* command on the terminal and open the app on the browser at the *http://localhost:3000/books*. You should see array of books objects displayed as follow:

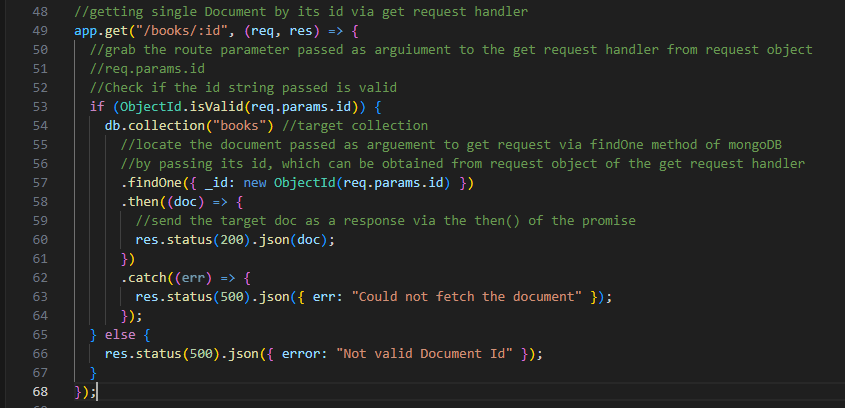


**18-Finding Single Document**

* Import the “ObjectId” from MongoDB to access the ID objectS mongo generated.

const {ObjectId}=require("mongodb");

* Reach for the target collection(books) we want to query for the document, grab the the ‘id’(can be any name) from the request object of get request handler, check if this id is valid MongoDB document id format. If it is valid pass this id to the findOne() method of MongoDB and send the target document as response. Else send error message to the user. Note, if the id is valid, but does not exist in the DB, we receive “null” as a response, so figure a way to handle this null when rendering to the user.



* Test it on the browser
  + Take the id of a book from the database and pass it in the address bar as “url”

http://localhost:3000/books/66ec68787709b1cc46c73bfd

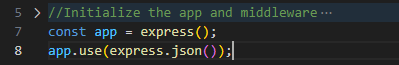
* + Note, if the ID is valid but it doesn’t exist in the database, you will get “null”. So you can handle this in server side or front end of you application.

**19 - Using POSTMAN**

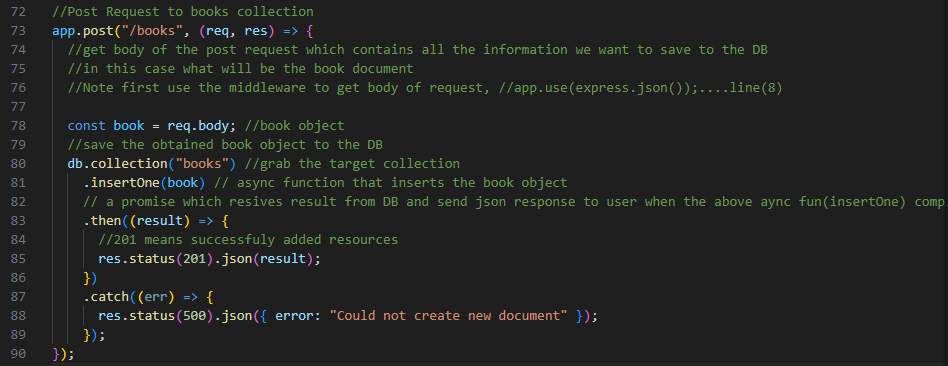
* Simulates request to the API we are building, and show us the response we get back from those requests, because it is difficult to perform “delete”, “post” and “Update” requests from browser.
* allow us to simulate the request and response actions and test them until you write front-end JavaScript code to do these request and responses.
* In addition, it allows us to save and organize all the requests we make.
* Installation:
  + Download postman from <https://www.postman.com/downloads/>
  + Follow the installation instructions
  + Sign for free account for firs use
* Using Postman:
  + Starting new request
    - Make sure the server is started by excusing the “*nodemon app*” command
    - Press “+” Sign on the tab (or File > New Tab )
    - Select request type such as “Get” from the dropdown menu, and front of it enter the URL for the request/ end point we want to access : <http://localhost:3000/books> and press send button to send the request, you should see the response at the bottom.
  + Saving and opening the request
    - You can save these requests for later use by clicking the save button, you can give a name for the request on the next page if you want.
    - In addition, you can give the collection name while saving to organize your requests at one place > Create > Save. This will create the collection on the left nav menu.
    - You can open this saved request link from the list on the left section (navigation menu) and click send button to send the request.

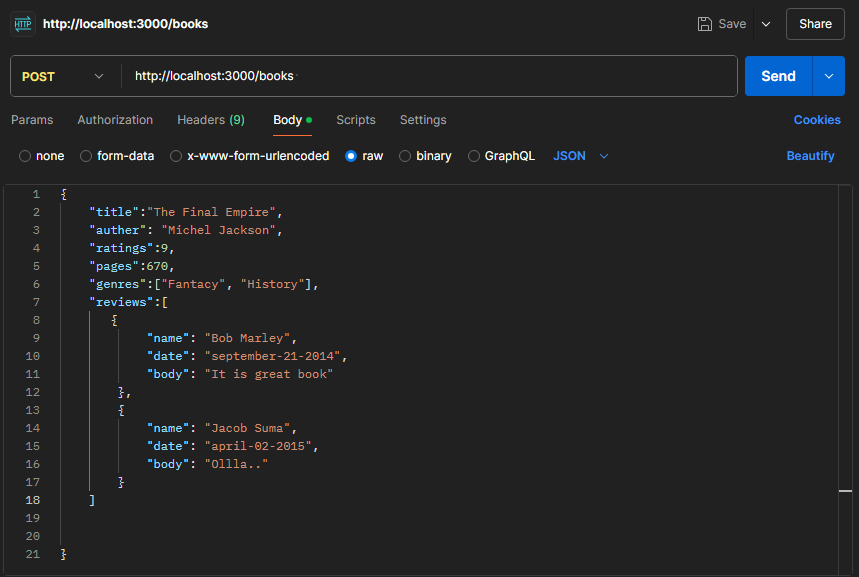
**20 - Handling POST Requests**

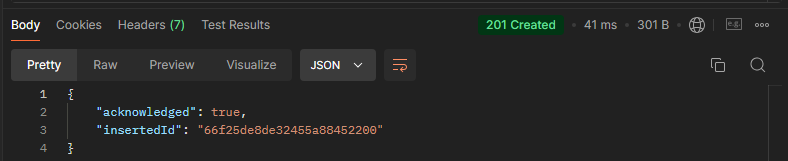
* To make a post request you will grab the body of the post request which contains all the information about the specific document we want to save to the DB.
* You need middleware provided by Express to get the body of the request via “req.body” property
  + utilize *use( )* method to use the middleware in our app which facilitate the access to the body of request. It takes *Json()* method as argument, which pass any “json” coming in with the request, so that we can use it inside our handler functions(like post request).

//only line 8

* + store the document data(book object we want to save to the DB) obtained via the middleware from the request body on a variable, then target the collection and invoke insertOne() method of mongoDB on it by passing the book object as argument to it.
  + Attach a promise, then() method which gets result from the database and sends Json response to the user, when the async function insertOne() is completed.
  + Finally add the *catch()* method which sends a response to user if there is error while adding the document via post request.



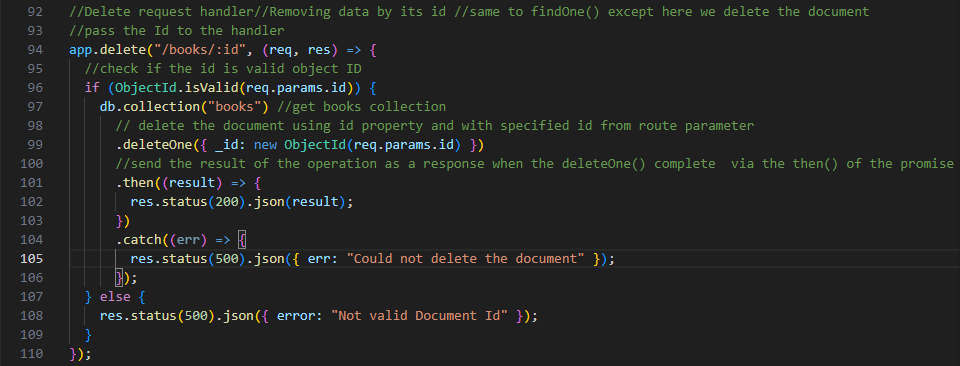
* Test it on postman
  + Create new request by Opening new tab on postman> choose “post” request from dropdown and add the request “url” Infront of it.( <http://localhost:3000/books> )
  + Choose the body tab to specify the book document we want save, and
  + check the “raw” radio button (or select from dropdown) and
  + select JSON from dropdown, for what type we send as request body
  + inset the book object you want to add to the collection(the request body):
* 
* Then press send button, you should get acknowledgment, along the insertion id:



* Then the post request on postman for later use

**21-Handling DELETE Requests**

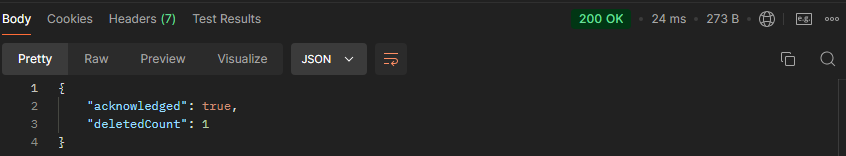
* Removing Specific data; is similar to find document we done earlier, you grab the document id form request object and check if the id is valid and it is valid delete it send the result of the operation as JSon, else send error message. Note, here we don’t get document like *findOne()* method. here unlike findOne() method we receive result instead of the document as a response , so we send back these result as a Json.



* Test it on the Postman
  + Create new Delete request: open new tab(+) > choose delete request from dropdown and on the text-field infant of it provide the URL for the book id we want to delete: <http://localhost:3000/books/66f25de8de32455a88452201>



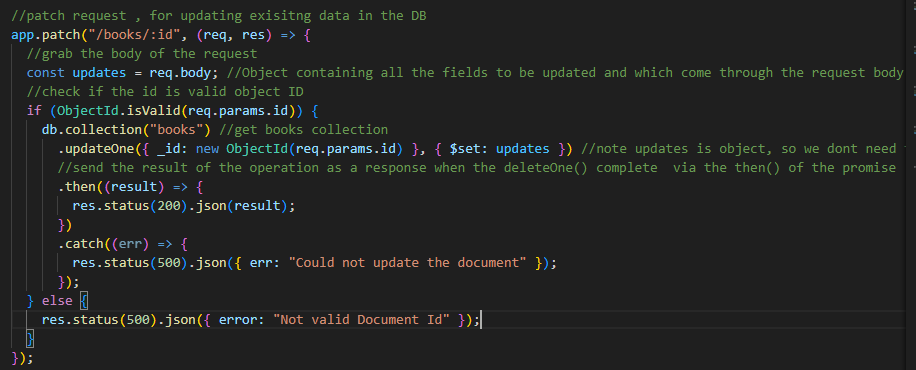
* + Press send, then you should get the acknowledgment of the deletion



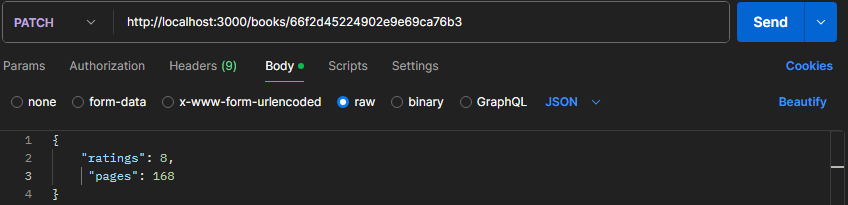
* + Then save the request for later use

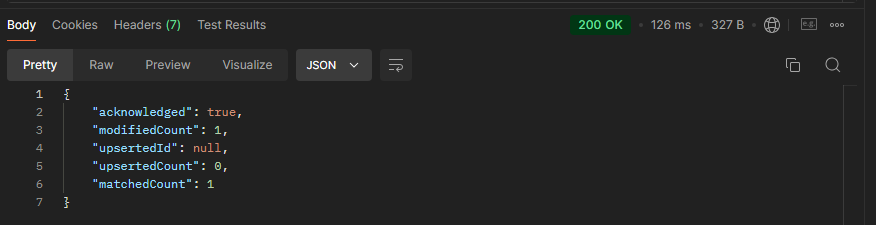
**22 - PATCH Requests**

* Patch request Updates the existing data in the DB. Similar to post request which sends data as body of the request which creates new entry, but here we don’t create new entry we modify exiting one by grabbing it by its ID. i.e. we send new field of the document as a body of the request and modifies values of existing document.
* First check if the received id from the request is valid, then get the collection and call the *updateOne()* method on it, which takes two argument( the first finds the document based on its id and the second set the update via “$set Operator).
* The fields which are not included with the request are left intact, but the ones which are specified are modified by this operation.



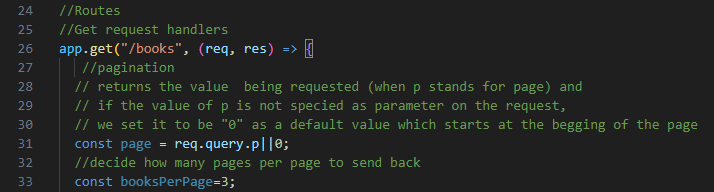
* Test it on postman
  + Create new patch request: new tab(+) > select patch request from dropdown and provide the URL to the document ID: <http://localhost:3000/books/66f2d45224902e9e69ca76b3>
  + Choose Body tab from the params list, and check raw data and select JSON as data format, then specify the fields you want to update as an object( for example lets update only the ratings and page number of the document specified by its id in the URL).



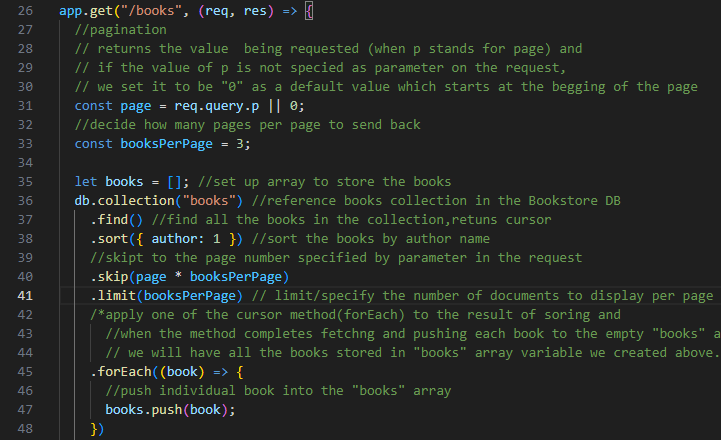
* + Then press send the send button for the updates to take effect, then you should see the acknowledgement as well on the window at the bottom.
  + 

**23 - Pagination**

* A way to retrieve and display get request data in batches, i.e. instead of displaying all of the documents at once, we use query parameters to retrieve documents page by page( eg. 20 docs/page). This can be done with our UI or frontend and button.
* You can get the query parameter for the pagination from the Express request object
  + Store the parameter passed to the request on variable and set default value(0-first page) incase user doesn’t provide query parameter for the page.
  + Decide number of documents you want to display per page(let say 3 for this example)



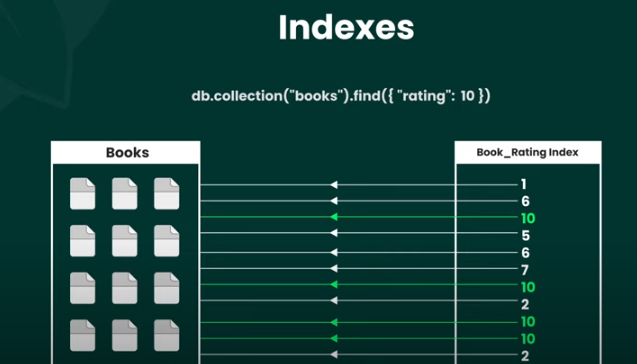
* + Use combination of *skip()* to skip certain amount of documents(say books) and *limit()* method to limit how many documents we get back.
    - For example, assume we have 3 books per page and we want to get the second page from returned result, so we use skip() to which skips the page number times books per page then fetch the next three books(use limt() to bring only 3 books).



* + Notice the line 32,33, 40 and 41( this line codes are added to get request route we set at the beginning to implement the pagination.
  + Test it on postman, if we can display only 3 books per page when we specify query parameter on the request
    - If you send the get request to the URL <http://localhost:3000/books> you should get only the first 3 book documents. Since we haven’t provided the parameter for page to display the paginator takes the default value “0” and displays the first page with 3 documents.
      * This is also equal to query set to 0 as follow: <http://localhost:3000/books?p=0>
    - If you set query to P=1( <http://localhost:3000/books?p=1> ), it will skip the first three books and displays the next 3 books on page 1

**24 - Indexes**

* Indexes allow the database server too perform specific queries to find documents much more efficiently without needing to examine an entire collection of documents.
* In MongoDB index is list of values of certain fields in our documents. For example, we can have index based on “ratings” field of the book documents in the books collections and each of those ratings in the index will have a pointer pointing back to the document it came from. Then when query comes into MongoDB based on this criteria(e.g. fetch all the documents with ratings of 10), MongoDB just need to look at the rating index and easily see every value of 10 and then find those corresponding documents inside the collection and return them to us.



* This makes the process much more efficient and quicker, since MongoDB doesn’t have to examine every single document. It just scans the index instead and then it finds the corresponding documents.
* For more on indexes and usage view the following video: <https://www.youtube.com/watch?v=D14wWW9EEx8&list=PL4cUxeGkcC9h77dJ-QJlwGlZlTd4ecZOA&index=24>

**25 - MongoDB Atlas**

* <https://www.youtube.com/watch?v=084rmLU1UgA&list=PL4cUxeGkcC9h77dJ-QJlwGlZlTd4ecZOA&index=25>