

```
1  /* Comparison Operators
2
3  ///? 1 $eq: Matches values that are equal to the specified value.
4  // db.products.find({'price': {$eq:699}})
5
6  ///? 2: $ne: Matches values that are not equal to the specified value.
7  // db.products.find({'price': {$ne:699}}).count()
8
9  // 3: $gt: Matches values that are greater than the specified value.
10 // db.products.find({'price': {$gt:699}})
11
12 // 4: $gte: Matches values that are greater than or equal to the specified value.
13 // db.products.find({'price': {$gte:699}})
14
15 // 5: $lt: Matches values that are less than the specified value.
16 // db.products.find({'price': {$lt:699}})
17
18 // 6: $lte: Matches values that are less than or equal to the specified value.
19 // Find products with price less than or equal to 699
20 // db.products.find({'price': {$lte:699}})
21
22 // $in: Matches values that are within the specified array.
23 // db.products.find({'price': 129, 'price':39})
24 // db.products.find({'price': {$in: [129,39]}})
25 ///? Now here I will go with different collection
26 // db.category.find({ name: { $in: ["Travel & Luggage", "Home & Kitchen"] } });
27
28 ///? $nin: Matches values that are not within the specified array.
29 // db.products.find({'price': {$nin: [249,129,39]}}).count()
30
31
32
33 // *****
34 /* Index
35 // *****
36 // lets query to see total count of products greater than 100:
37 // db.products.find({ price: { $gt: 100 } }).count();
38
39 ///? We can use explain() method to understand it more better
40 ///? Also we can add explain('executionStats') to understand more in depth
41
42 ///? Find name= air fryer from the products collections
43 // db.products.explain('executionStats').find({'name':'Air Fryer'})
44 ///! executionTimeMillis: 18,
45
46 /* Creating Indexes
47 ///? Indexes can be created using the createIndex() method.
48 ///? syntax: db.collectionName.createIndex({ fieldName: 1 });
49 ///? In this case, 1 represents ascending order, and -1 would be descending order.
50
51 // db.products.createIndex({name:1})
52 ///! executionTimeMillis: 8
53
54 /* Getting Indexes
55 // db.products.getIndexes();
56 ///?Did you realize that _id is already there? _id is automatically added by mongodb
  and it's a default unique index.
57
58 /* Removing an index
```

```
59 // db.products.dropIndex({ name: 1 });
60
61 /* Creating a unique index
62 // db.users.createIndex({ email: 1 }, { unique: true });
63 /* When not to use indexes?
64 // Indexes can actually slow things down in some conditions, it usually slows things
   down if your query is going to return huge amounts of data. It's unnecessary to use
   indexes everywhere.
65
66 // For example
67 // db.products.explain('executionStats').find({price: {$gt: 100}});
68 // The think it is returning almost 90% of data in output
69 // nReturned: 9216,
70
71 // let create the index for the Price
72 // db.products.createIndex({price:1})
73 // Now lets check the time its taking
74 // executionTimeMillis: 26, it is taking almost double time
75
76
77
78 /* Cursor Methods
79 // todo We need to use () parenthesis in all the cursor methods
80 /*? 1: count(): The count() method returns the count of documents that match the
   query.
81 // db.products.find({price: {$gt: 250}}).count()
82 /*? 2: limit(): The limit() method restricts the number of documents returned by the
   query.
83 // db.products.find({price: {$gt: 250}}).limit(2)
84 /*? 3: skip(): The skip() method skips a specified number of documents and returns
   the rest.
85 // db.products.find({price: {$gt: 250}}).limit(5).skip(1)
86 /*? 4: sort(): The sort() method sorts the documents based on the specified
   field(s).
87 // db.products.find({price: {$gt: 1250}}).limit(3).sort({'price':1})
88
89
90
91 //TODO We will work with multiple data and it will be fun and it is going to be the
   best video over the internet for sure
92 /*? 1: Creating / Deleting Databases
93 // show dbs
94 // use thapaProducts
95 // db.dropDatabase()
96 /*? 2: Creating / Deleting Databases and Collections
97 // db.createCollection('test')
98 // show collections
99 // db.test.drop() // where the test is the collection name
100
101
102
103 /* Elements Operator
104 // In MongoDB, element operators are used to query documents based on the existence,
   type, and values of fields within the documents. These operators help you work with
   fields that are arrays, null, missing, or have specific data types.
105 /*? 1: $exists: Matches documents that have a specific field, regardless of its
   value.
106 // db.products.find({ price: { $exists: true } }).count();
107 // Find documents with the "price" field present, and if it's present then check the
   value greater then 1200
```

```
108 // db.products.find({ price: { $exists: true },cls price: { $gt: 1250 } });
109 //? 2: $type: The $type operator filters documents based on the BSON data type of a
    field.
110 // Basically we need to search or find the fields based on types (BSON Type) for
    example
111 // db.products.find({ price: { $type: "string" } });
112 // result will be 0, bcz the price type is number
113 // db.products.find({ price: { $type: "number" } }).count()
114 // 1: Double
115 // 2: String
116 // 3: Object
117 // 4: Array
118 // 5: Binary data
119 // 6: Undefined
120 // 7: Object id
121 // 8: Boolean
122 // 9: Date
123 // 10: Null
124 // 11: Regular expression
125 // 12: JavaScript code
126 // 13: Symbol
127 // 14: JavaScript code with scope
128 // 17: 64-bit integer
129 // db.products.find({ price: { $type: "string" } });
130 // result will be 0, bcz the price type is number
131 // db.products.find({ price: { $type: "number" } }).count()
132 //? 3: $size: The $size operator matches documents where the size of an array field
    matches a specified value.
133 // db.comments.find({comments: {$size:2}})
134
135
136
137 /** Embedded Documents (Dealing with Arrays & Object)
138 /** Just use the dot notations, that's it
139 // ?1: Find posts with comments by a specific user (Array)
140 // db.comments.find({'comments.user': 'Alice'})
141 //? 2: Find the documents where the views in metadata field > 1200. (Object)
142 // db.comments.find({ "metadata.views": { $gt: 1200 } });
143 //? 3: we need to find out the document where the user in comments = Henry and also
    the in the metadata likes value > 50.
144 // db.comments.find({ 'comments.user':'Henry' , 'metadata.likes':{$gt: 50} })
145 //? 4:we need to return an comments array which must have this two (user = alice &
    vinod) elements only in it.
146 //! We need to use $all operator. Here the order doesn't matter.
147 // db.comments.find({ "comments.user": { $all: ["Alice", "Vinod"] } });
148 // db.comments.find({'comments.user': {$all: ['Alice','Vinod','Bob']}})
149 //? 5: In Array for multiple querying we user $elemMatch operator.
150 /** Here is the syntax: {field: {$elemMatch: { {query1},{query2}.. } }}
151 // db.comments.find({ comments: {$elemMatch: {'user':'Alice' , 'text':'Awesome
    article!' } } })
152 //? by the wat you can write the same using simple way
153 // db.comments.find({ 'comments.user':'Alice' , 'comments.text':'Awesome article!'
    } )
154
155
156
157 /** Introduction to $expresion
158 // The $expr operator in MongoDB allows you to use aggregation expressions within a
    query to compare fields from the same document. It's particularly useful when you
    need to perform more complex comparisons or calculations involving document fields.
```

```

159 //? The syntax is {$expr: {operator: [field, value] } }
160 // One important thing to remember is the field should be prefix with $ sign.
161 // db.products.find({'$expr': {'$gt': ['$price',1340] }})
162 //! Find sales where (quantity * price) is greater than targetPrice
163 // db.sales.find({
164 //     $expr: {
165 //         $gt: [{ $multiply: ['$quantity', '$price'] }, '$targetPrice'],
166 //     },
167 // });
168 // here both the values are fields only for comparison thats why $ sign is used
169
170
171 /* Logical Operators
172 //We have 4 logical operators with us. $and , $or, $nor and $not
173 // syntax: { operator: [{condition1},{condition2},...] }
174 //? 1: $and: Performs a logical AND operation on an array of expressions, where all
    expressions must be true for the document to match.
175 //! Find products with price greater than 100 and name equal to "Diamond Ring"
176 // db.products.find({ $and: [ { 'price': { $gt: 10 } }, { 'name': 'Notebook
    Collection' } ] })
177 // db.products.find({'price': {'$gt':10}, 'name':{'$eq': 'Notebook Collection'}})
178 /* In MongoDB, when you provide multiple fields within a single query document,
    MongoDB treats them as an implicit AND operation.
179 //? 2: $or: Performs a logical OR operation on an array of expressions, where at
    least one expression must be true for the document to match.
180 // We can use logical operator only when we have the duplicate fields
181 // db.products.find({'price': 129, 'price':39})
182 // but we can write the same in $or operator
183 // db.products.find({ $or: [{ price: 129 }, { price: 39 }] });
184 //? 3: $not: Performs a logical NOT operation on the specified expression, inverting
    the result.
185 //? Find products with price not equal to 100
186 // db.products.find( {'price': {'$not': {'$eq': 100}} } )
187 //?4: $nor: Performs a logical NOR operation on an array of expressions, where none
    of the expressions must be true for the document to match
188 /* Find products with price not equal to 100 or name not equal to "Notebook
    Collection"
189 // db.products.find( {$nor: [ {'price': {'$eq': 100}}, {'name':'Notebook Collection'
    }] } )
190
191
192 /* MONGOIMPORT
193 // Now I will show How to import data from json file
194 //? mongoimport E:\\mongo\\products.json -d shop -c products
195 //? mongoimport E:\\mongo\\products.json -d shop -c products --jsonArray
196
197
198 /* Projection
199 // Which filed to display which not, only the _id is needs to be explicitly defined
    so that the _id won't be included in our output.
200 // Including Specific Fields: To include only specific fields in the query result,
    you can use the projection with a value of 1 for the fields you want to include.
201 // db.products.find({}, { name:1, price:1}).limit(2)
202 // Excluding Specific Fields:To exclude specific fields from the query result, you
    can use the projection with a value of 0 for the fields you want to exclude.
203 // db.products.find({}, {_id:0, name:1, price:1}).limit(2)
204 //! We cannot include and exclude fields in the same query projection in MongoDB.
    It's either inclusion or exclusion, not both simultaneously.
205 // db.products.find({}, { _id: 0, name: 1, price: 1, price: 0 }).limit(2);
206

```

```
207
208 // Aggregations Example 🗨 //
209 // *****
210 /* Aggregation Framework
211 // *****
212
213 // The Aggregation Framework is a powerful feature in MongoDB that allows you to
    process and analyze data in a highly flexible and efficient manner. It provides a
    set of pipeline stages that enable you to perform data transformations, group data,
    and perform various calculations on collections.
214
215 // In MongoDB's aggregation framework, $match, $group, and $unwind are referred to
    as aggregation operators. They are used as stages in the aggregation pipeline to
    perform specific operations on the data.
216
217 /* Aggregation Operations
218 /*? $match
219 /*? The $match stage filters documents based on specified conditions.
220
221 /*? Retrieve all products with a name = Sleek Wooden Tuna.
222
223 // db.products.aggregate([
224 //     {
225 //         $match: {
226 //             'name': 'Sleek Wooden Tuna'
227 //         }
228 //     }
229 // ])
230
231 /*? Retrieve all products with a price greater than 50.
232 // db.products.aggregate([
233 //     { $match: { price: { $gt: 50 } } }
234 // ]]);
235
236
237 ❤ Thank You So Much For Choosing My Video ❤
238
239 Hi everyone,
240
241 I'm absolutely thrilled - we're almost at 600K subscribers for our MongoDB course!
    This course was a true labor of love, and it's been amazing to see how it's helping
    you all.
242
243 If you've enjoyed what we're doing and want to be part of our journey, hitting that
    Subscribe button would mean the world to me. Let's keep growing and learning
    together!
244 Here is the link: https://www.youtube.com/thapatechnical
245
246 With gratitude,
247 Thapa Technical
248
249 /* $group
250 // The $group stage groups documents by specified fields and performs aggregation
    functions. it is like the reduce methods in JS
251
252 // when dealing with $group stage we need to pass $sign for our existing field not
    the one we are going to create
253 // syntax :
254 // {
255 //     $group:
```

```
256 //      {
257 //          _id: <expression>, // Group key
258 //          <field1>: { <accumulator1> : <expression1> },
259 //          ...
260 //      }
261 //  }
262
263 //
264 https://www.mongodb.com/docs/v6.0/reference/operator/aggregation/group/#consideratio
265 ns
266 db.products.aggregate([
267   { $match: { price: { $gt: 900 } } },
268   {
269     $group: {
270       _id: { sameCompany: "$company" },
271       totalPrice: { $sum: "$price" },
272     },
273   },
274 ]);
275 // let's use another accumulator operations
276 // $avg
277
278 // find the quantity = 5, group them with same quantity and find the average price
279
280 db.sales.aggregate([ { $match: { quantity: { $eq: 5 } } } ]]);
281 // both are same
282 db.sales.aggregate([
283   { $match: { quantity: 5 } },
284   {
285     $group: {
286       _id: { quan: "$quantity" },
287       avgPrice: { $avg: "$price" },
288     },
289   },
290 ]);
291
292 /* $sort
293
294 db.products.aggregate([
295   { $match: { price: { $gt: 1200 } } },
296   {
297     $group: {
298       _id: "$category",
299       totalPrice: { $sum: "$price" },
300     },
301   },
302   { $sort: { totalPrice: 1 } },
303 ]);
304
305 // $sort is like .sort() but you can even sort the values that you added in group.
306 // (Of course you can also sort before grouping or with any other values. But here you
307 // can even sort in ascending or descending based on number of products it has.
308
309 db.products.aggregate([
310   { $match: { price: { $gt: 1200 } } },
311   {
312     $group: {
313       _id: "$category",
```

```
312     totalPrice: { $sum: "$price" },
313   },
314 },
315 { $sort: { totalPrice: -1 } },
316 ]));
317
318 /* $project
319
320 db.products.aggregate([
321   {
322     $project: {
323       _id: 0,
324       price: 1,
325       name: 1,
326     },
327   },
328 ]));
329
330 // We can use the $project stage to create new fields by applying expressions or
transformations to existing fields. For example, you could calculate the discounted
price as a new field:
331
332 db.products.aggregate([
333   { $match: { price: { $gt: 1000 } } },
334   {
335     $project: {
336       _id: 0,
337       name: 1,
338       originalPrice: "$price",
339       disPrice: { $multiply: ["$price", 0.8] },
340     },
341   },
342 ]));
343
344 // again we can add the sort here too
345
346 db.products.aggregate([
347   { $match: { price: { $gt: 1000 } } },
348   {
349     $project: {
350       _id: 0,
351       name: 1,
352       originalPrice: "$price",
353       disPrice: { $multiply: ["$price", 0.8] },
354     },
355   },
356   { $sort: { disPrice: -1 } },
357 ]));
358
359 /* $push and $unwind
360 ///? Find documents with a price greater than 1200, then group them by price and
create an array of colors for each group.
361
362 /* Before
363 /* if price = 1250 => colors: [ '#000000', '#cc6600', '#663300' ],
364 /* if price = 1250 => colors: [ '#fff000', '#dddd', '#663300' ],
365
366 ///? After, I need a new document where
367 {
368   price: 1250,
```



```
369   allColors: ['#000000', '#cc6600', '#663300', '#fff000', '#dddd', '#663300' ]
370 }
371
372 // code
373 db.products.aggregate([
374   { $match: { price: { $gt: 1200 } } },
375   {
376     $group: {
377       _id: { priceGroup: "$price" },
378       colors: { $push: "$colors" },
379     },
380   },
381 ]);
382
383 /* $unwind
384
385 db.products.aggregate([
386   { $match: { price: { $gt: 1200 } } },
387   { $unwind: "$colors" },
388   {
389     $group: {
390       _id: { priceGroup: "$price" },
391       colors: { $push: "$colors" },
392     },
393   },
394 ]);
395
396 //? Before
397 {
398   _id: ObjectId("64c23601e32f4a51b19b9263"),
399   name: 'Laptop Pro',
400   company: '64c23350e32f4a51b19b9231',
401   price: 1299,
402   colors: [ '#333333', '#cccccc', '#00ff00' ],
403   image: '/images/product-laptop.png',
404   category: '64c2342de32f4a51b19b924e',
405   isFeatured: true
406 },
407
408 //! $unwind: '$colors';
409 //? the $unwind stage deconstructs the "colors" array, creating multiple documents
  for each color within a product.
410
411 //? After
412 {
413   _id: ObjectId("64c23601e32f4a51b19b9263"),
414   name: 'Laptop Pro',
415   company: '64c23350e32f4a51b19b9231',
416   price: 1299,
417   colors: '#333333',
418   image: '/images/product-laptop.png',
419   category: '64c2342de32f4a51b19b924e',
420   isFeatured: true
421 },
422
423 // {
424 //   _id: ObjectId("64c23601e32f4a51b19b9263"),
425 //   name: 'Laptop Pro',
426 //   company: '64c23350e32f4a51b19b9231',
427 //   price: 1299,
```



```
428 //   colors: '#cccccc',
429 //   image: '/images/product-laptop.png',
430 //   category: '64c2342de32f4a51b19b924e',
431 //   isFeatured: true
432 // },
433
434 // so now all the colors are in a string format, so $push will add them as an
  element in an array of colors
435
436 db.products.aggregate([
437   { $match: { price: { $gt: 1200 } } },
438   { $unwind: "$colors" },
439   {
440     $group: {
441       _id: null,
442       totalCount: { $sum: 1 },
443     },
444   },
445 ]);
446
447 db.products.aggregate([
448   { $match: { price: { $gt: 1200 } } },
449   { $unwind: "$colors" },
450   {
451     $group: {
452       _id: { priceGroup: "$price" },
453       colors: { $push: "$colors" },
454     },
455   },
456 ]);
457
458 /* $addToSet
459 // still there is a problem and that is we are also getting the duplicates values so
  to remove it we will use the $addToSet
460
461 db.products.aggregate([
462   { $match: { price: { $gt: 1200 } } },
463   { $unwind: "$colors" },
464   {
465     $group: {
466       _id: { priceGroup: "$price" },
467       colors: { $addToSet: "$colors" },
468     },
469   },
470 ]);
471
472 /* $size
473 // What If we want to count the number of unique colors for each price group
474 db.products.aggregate([
475   { $match: { price: { $gt: 1200 } } },
476   { $unwind: "$colors" },
477   {
478     $group: {
479       _id: { priceGroup: "$price" },
480       colors: { $addToSet: "$colors" },
481       colorLength: { $size: "$colors" },
482     },
483   },
484 ]);
485
```

```
486 // we can't do this, bcz the $size operator is not allowed directly within the
    $group stage. Instead, you can use it in combination with other aggregation
    operators or in separate pipeline stages.
487
488 db.products.aggregate([
489   { $match: { price: { $gt: 1200 } } },
490   { $unwind: "$colors" },
491   {
492     $group: {
493       _id: { priceGroup: "$price" },
494       allColors: { $addToSet: "$colors" },
495     },
496   },
497   {
498     $project: {
499       _id: 1,
500       allColors: 1,
501       colorLength: { $size: "$allColors" },
502     },
503   },
504   { $limit: 1 },
505 ]);
506
507 /*! very Important in project stage we are only getting two fields and the name of
    the fields has to match with the fields names in group stage. ex. allColors fields
508
509 /* limit
510
511 db.products.aggregate([
512   { $match: { price: { $gt: 1200 } } },
513   { $unwind: "$colors" },
514   {
515     $group: {
516       _id: { priceGroup: "$price" },
517       allColors: { $addToSet: "$colors" },
518     },
519   },
520   {
521     $project: {
522       _id: 1,
523       allColors: 1,
524       colorLength: { $size: "$allColors" },
525     },
526   },
527   { $limit: 1 },
528 ]);
529
530 /* skip
531
532 db.products.aggregate([
533   { $match: { price: { $gt: 1200 } } },
534   { $unwind: "$colors" },
535   {
536     $group: {
537       _id: { priceGroup: "$price" },
538       allColors: { $addToSet: "$colors" },
539     },
540   },
541   {
542     $project: {
```

```

543     _id: 1,
544     allColors: 1,
545     colorLength: { $size: "$allColors" },
546   },
547 },
548 { $skip: 1 },
549 ]));
550
551 /* $filter
552 db.col.aggregate([
553   {
554     $project: {
555       name: 1,
556       values: {
557         $filter: {
558           input: "$values",
559           as: "value",
560           cond: { $gt: ["$$value", 30] },
561         },
562       },
563     },
564   },
565 ]));
566
567 // Or Example 🗨 //
568 db.produts.aggregate([
569   {
570     $group: {
571       _id: "$company",
572       totalProducts: { $sum: "$price" },
573     },
574   },
575 ]));
576
577 db.products.aggregate([
578   {
579     $match: {
580       _id: "64c23350e32f4a51b19b9247",
581     },
582   },
583 ]));
584
585
586 price > 900
587 company $group
588 sum price
589
590 db.produts.aggregate([
591   {
592     $match: {price: {$gt: 900}}
593   },
594   {
595     $group: {
596       _id: "$company",
597       totalProducts: { $sum: "$price" },
598     },
599   },
600 ]));
601

```

```

602  #!/ find the quantity = 5, group them with same quantity and find the average
603  price
604  db.sales.aggregate([
605    { $match: {quantity:5} },
606    {
607      $group: {
608        _id: '$quantity',
609        priceTotal: {$sum: '$price'},
610        pricrAvg: {$avg: '$price'}
611      }
612    }
613  ])
614
615  db.products.aggregate([
616    { $match: { price: { $gt: 1200 } } },
617    {
618      $group: {
619        _id: "$category",
620        totalPrice: { $sum: "$price" },
621      },
622    },
623    {
624      $sort: {totalPrice: 1}
625    }
626  ]);
627
628
629  db.products.aggregate([
630    { $match: { price: { $gt: 1200 } } },
631    {
632      $project: {
633        price:1,
634        discountPrice: {$multiply: ['$price', 0.8]}
635      }
636    }
637  ])
638
639  db.products.aggregate([
640    { $match: { price: { $gt: 1200 } } },
641    {
642      $group: {
643        _id: '$price',
644        allColors: { $push : '$colors' }
645      }
646    }
647  ])
648
649  price: 1999,
650  colors: [ '#000000', '#cc6600', '#663300' ]
651
652  price: 1999,
653  colors: [ '#000000', '#cc6600', '#663300' ]
654
655
656  price: 1999,
657  colors: [
658    [ '#000000', '#cc6600', '#663300' ],
659    [ '#000000', '#cc6600', '#663300' ]
660  ]

```

```
661 ,
662 price: 1999,
663 colors: ['#000000', '#cc6600', '#663300']
664
665 db.products.aggregate([
666   { $unwind: '$colors' },
667   { $match: { price: { $gt: 1200 } } },
668   {
669     $group: {
670       _id: '$price',
671       allColors: { $push : '$colors' }
672     }
673   }
674 ])
675
676 db.products.aggregate([
677   { $unwind: '$colors' },
678   { $match: { price: { $gt: 1200 } } },
679   {
680     $group: {
681       _id: '$price',
682       allColors: { $addToSet : '$colors' }
683     }
684   }
685 ])
686
687 db.products.aggregate([
688   { $match: { price: { $gt: 1200 } } },
689   { $unwind: "$colors" },
690   {
691     $group: {
692       _id: { priceGroup: "$price" },
693       colors: { $addToSet: "$colors" },
694     },
695   },
696   {
697     $project: {
698       _id: 1,
699       colors: 1,
700       colorLength: { $size: "$colors" },
701     }
702   },
703   {
704     $limit: 1
705   }
706 ]);
707
708
709 db.col.insertMany([
710   {
711     _id: "64c23350e32f4a51b19b9201",
712     name: "Document 1",
713     values: [10, 20, 30, 40, 50],
714   },
715   {
716     _id: "64c23350e32f4a51b19b9202",
717     name: "Document 2",
718     values: [15, 25, 35, 45, 55],
719   },
720   {
```

```

721     _id: "64c23350e32f4a51b19b9203",
722     name: "Document 3",
723     values: [5, 15, 25, 35, 45],
724 },
725 {
726     _id: "64c23350e32f4a51b19b9204",
727     name: "Document 4",
728     values: [30, 40, 50, 60, 70],
729 },
730 {
731     _id: "64c23350e32f4a51b19b9205",
732     name: "Document 5",
733     values: [25, 35, 45, 55, 65],
734 },
735 ]
736 )
737
738
739 db.col.aggregate([
740     $project: {
741         name: 1,
742         thapaValue: {
743             $filter: {
744                 input: '$values',
745                 as: 'val',
746                 cond: { $gt: ['$val', 30] }
747             }
748         }
749     }
750 ]])
751
752 // // CRUD Example  OPERATIONS IN MONGODB 📌 // //
753 // ** inserts Operation ** //
754
755 ///? 3: crete (Inserting the documents in collection)
756 ///!methods like insert() and save() are being deprecated in favor of more explicit
757 methods like insertOne() and insertMany()
758
759 ///? 1.a insertOne(): This method inserts a single document into the collection.
760 // db.product.insertOne({ name: "vinod", age: 29 });
761
762 ///? 1.b insertMany(): This method inserts an array of documents into the collection.
763 ///! IMP - Argument "docs" must be an array of documents (use array always)
764 //db.product.insertMany([{ 'name': 'vinod', 'age': 29 }, { 'name': 'arjun', 'age': 30 }])
765
766 ///? Important when to use quotes and when to not
767 ///Special Characters: If your field name contains special characters, spaces, or
768 starts with a numeric digit, using quotes becomes necessary.
769 // Field name with spaces
770 // db.grades.find({"course name": "Math"})
771 // Field name starting with a digit
772 // db.grades.find({"1st_place": true})
773
774 ///Reserved Words: If your field name is a reserved keyword in MongoDB (e.g., $group,
775 $sum, etc.), you need to use quotes to distinguish it from the reserved keyword
776 // we will see when we will see comparison operator
777
778
779 ///? 1.c Ordered Inserts vs Unordered Inserts
780 ///In MongoDB, "ordered" and "unordered" refer to the behavior of a bulk write
781 operation when multiple operations are included in a single batch. {ordered:1 or -1}

```

```

By default it's true. If any individual operation fails, MongoDB stops processing
further operations in the batch and returns an error.
777
778 //? it's a example of ordered Inserts after the 2nd execution it will stop
779 // db.product.insertMany([
780 //   { name: "vinod", age: 29 },
781 //   { _id: ObjectId("64cb3ea5be4cb31d576182a3"), name: "sujan" },
782 //   { name: "naran", age: "30" },
783 // ]);
784
785 //? Unordered Inserts
786 // db.product.insertMany(
787 //   [
788 //     { name: "vinod", age: 29 },
789 //     { _id: ObjectId("64cb3ea5be4cb31d576182a3"), name: "sujan" },
790 //     { name: "ram", age: "30" },
791 //   ],
792 //   { ordered: false }
793 // );
794 //In this example, even though the 2nd operation fails due to the duplicate _id,
MongoDB continues processing and returns a result object with information about both
successful and failed operations.
795
796 //? 3: Case Sensitivity in MongoDB
797 //In MongoDB, collection names are case-sensitive. Therefore, db.Product and
db.product are considered as two different collections. The same rule applies to
field names within documents.
798 // db.Product.insertOne({name:"thapa",age:30})
799 // vs
800 // db.product.insertOne({name:"thapa",age:30})
801 // the output will be two collections 🤖
802 // dbproduct> show collections
803 // product
804 // Product
805
806
807 // ** Read Operations ** //
808
809 /* 2.a find(): The find() method is the most common way to retrieve documents from
a collection. It allows you to specify query conditions to filter the documents you
want to retrieve.
810 //? syntax => db.collection_name.find({key:value})
811 // db.product.find({ name: "vinod" });
812 // db.product.find({'age':29})
813
814 /* 2.b findOne(): The findOne() method returns a single document that matches the
specified query condition. It's useful when you only need to retrieve one document.
815 //? syntax => db.collection_name.findOne({key:value})
816 // db.product.findOne({ age: 29 });
817 // db.product.findOne({'name':'vinod'})
818
819 /* MONGOIMPORT
820 // Now I will show How to import data from json file
821 //? mongoimport E:\\mongo\\products.json -d shop -c products
822
823 // mongoimport E:\\mongo\\products.json -d shop -c products --jsonArray
824
825 //! Failed: error reading separator after document #1: bad JSON array format - found
no opening bracket '[' in input source
826

```



```
827 // mongoimport E:\mongo\mongo_json\sales.json -d tshop -c sales --jsonArray
828
829 // mongoexport -c sales -d shop -o E:\mongo\sales1.json
830
831 // mongoexport --collection=sales --db=shop -out=E:\mongo\sales.json
832
833
834 // ** Delete Operations ** //
835 //? In MongoDB, the DELETE operations are used to remove documents from a
collection. There are two main methods to perform deletion: deleteOne() and
deleteMany().
836
837 /** Delete a Single Document:
838 //? syntax : db.collectionName.deleteOne({ _id: ObjectId("12345") });
839 // db.sales.deleteOne({ _id: 1 });
840
841 /** Delete Multiple Documents:
842 //? Syntax: db.collectionName.deleteMany({ field: "value" });
843 // db.sales.deleteMany({'price':55})
844
845
846 // ** Update Operations ** //
847 //? 1: Updating a Single Field:
848 /** db.collectionName.updateOne(
849 //   { _id: ObjectId("12345") },
850 //   { $set: { fieldName: "new value" } }
851 // );
852
853 //? Update the price value = 45 in a products collections, where the _id =
ObjectId("64c2363be32f4a51b19b9271")
854
855 //? Update the isFeatures value = true in a products collections, where the name =
Designer Handbag
856
857 /** UpdateMany
858 //? Update all the isFeatures value = true in a products collections, where the
price = 120
859
860 /** Updating multiple fields in a document
861 // db.collectionName.updateOne(
862 //   { _id: ObjectId("12345") },
863 //   {
864 //     $set: {
865 //       field1: "new value 1",
866 //       field2: "new value 2",
867 //     },
868 //   }
869 // );
870
871 //? Update the price = 154 and isFeatures = false fields from the products
collections where the name = Unbranded Frozen Chicken.
872
873 /** Renaming a field in a document.
874 // syntax: db.collectionName.updateOne(
875 //   { _id: ObjectId("12345") },
876 //   { $rename: { oldFieldName: "newFieldName" } }
877 // );
878 //? Rename the products collection isFeatured field to isFeature, where the price =
123
879
```

```
880 /** Adding a new field in a document
881 // db.collectionName.updateOne(
882 //     { _id: ObjectId("12345") },
883 //     { $set: { newField: "new value" } }
884 // );
885
886 /** Removing or Deleting the Field in a document
887 // To remove a field from documents in MongoDB, you can use the $unset update
operator.
888 // db.collectionName.updateOne(
889 //     { _id: ObjectId("12345") },
890 //     { $unset: { fieldName: 1 } }
891 // );
892
893 /** Update Embedded Documents
894 /**? How do you add a new element to an array using the $push operator?
895 // db.collectionName.updateOne(
896 //     { _id: ObjectId("12345") },
897 //     { $push: { arrayField: "new element" } }
898 // );
899
900 /**? Popping from an Array: Removing the last element from an array in a document.
901 // Syntax: db.collectionName.updateOne(
902 //     { _id: ObjectId("12345") },
903 //     { $pop: { arrayField: 1 } }
904 // );
905
906 /**? Updating a field within an embedded document.
907
908 /**? Update the text value within an comments array = "Awesome article!", where the
id=7 & username=alice.
909
910 // Consider this part of the query: 'comments.$.text': 'Awesome Thapa!'
911
912 // comments is the name of the array field.
913 // $ is the positional operator, and it refers to the index of the array element
that matches the query condition.
914 // text is the field within the specific comment element that you want to update.
915
916
917 // **** Monodb Opration Relation **** //
918 // const { MongoClient } = require("mongodb");
919 // const uri = "mongodb://127.0.0.1";
920 // const client = new MongoClient(uri);
921
922 // const data1 = {
923 //     name: "Designer Handbag1",
924 //     company: "64c23350e32f4a51b19b923a",
925 //     price: 3466,
926 //     colors: ["#000000", "#cc6600", "#663300"],
927 //     image: "/images/product-handbag.png",
928 //     category: "64c2342de32f4a51b19b9250",
929 //     isFeatured: true,
930 // };
931
932 // const main = async () => {
933 //     await client.connect();
934 //     const db = client.db("shop");
935 //     const collection = db.collection("products");
936
```

```
937 //   await collection.insertOne(data1);
938
939 //   const data = await collection.find({ price: { $eq: 3466 } }).toArray();
940 //   console.log(data);
941 //   return "done";
942 // };
943
944 // main()
945 //   .then(console.log())
946 //   .catch((e) => console.log(e))
947 //   .finally(() => client.close());
948
949
950
951 // **** Mongoose to Relation Operation **** //
952 const mongoose = require("mongoose");
953
954 // const uri = "mongodb://127.0.0.1/shop";
955
956 const uri =
957   "mongodb+srv://vbthapa55:qwerty123@cluster0.kziyfm.mongodb.net/shop?
  retryWrites=true&w=majority";
958
959 mongoose.connect(uri);
960
961 // we need to create a schema
962 const productSchema = new mongoose.Schema({
963   name: String,
964   company: String,
965   price: Number,
966   colors: [String],
967   image: String,
968   category: String,
969   isFeatured: Boolean,
970 });
971
972 // we need to now create an model
973 const Product = new mongoose.model("Product", productSchema);
974
975
976 // ❤️ Thank You So Much For Choosing My Video ❤️
977
978 // Hi everyone,
979
980 // I'm absolutely thrilled - we're almost at 600K subscribers for our MongoDB
  course! This course was a true labor of love, and it's been amazing to see how it's
  helping you all.
981
982 // If you've enjoyed what we're doing and want to be part of our journey, hitting
  that Subscribe button would mean the world to me. Let's keep growing and learning
  together!
983 // Here is the link: https://www.youtube.com/thapatechnical
984
985 // With gratitude,
986 // Thapa Technical
987
988
989
990 //? 2nd step while inserting the data
991 const data1 = {
```

```
992   name: "Designer Handbag2",
993   company: "64c23350e32f4a51b19b923a",
994   price: 2466,
995   colors: ["#000000", "#cc6600", "#663300"],
996   image: "/images/product-handbag.png",
997   category: "64c2342de32f4a51b19b9250",
998   isFeatured: true,
999 };
1000
1001 const main = async () => {
1002   try {
1003     //? 2: insert documents
1004     // await Product.insertMany(data1);
1005     // const data = await Product.find({ price: { $eq: 2466 } });
1006     // console.log(data);
1007
1008     //? 3 update query
1009     // await Product.findOneAndUpdate(
1010     //   { name: "Designer Handbag2", price: 2466 },
1011     //   { $set: { price: 2499 } }
1012     // );
1013
1014     //? 3 Delete query
1015
1016     await Product.findOneAndDelete({ name: "Designer Handbag2", price: 2499 });
1017     const data = await Product.find({
1018       name: "Designer Handbag2",
1019       price: 2499,
1020     });
1021
1022     console.log(data);
1023   } catch (error) {
1024     console.log(error);
1025   } finally {
1026     mongoose.connection.close();
1027   }
1028 };
1029
1030 main();
1031
1032 db.Students.insertMany(
1033   [{
1034     name: "Binamra",
1035     age: 20,
1036   },
1037   {
1038     name: "Thapa",
1039     age: 21,
1040   }
1041 );
1042
1043 {
1044   "name": "Binamra",
1045   "age": 20,
1046 }
1047 {
1048   "name": "Thapa",
1049   "age": 21,
1050 }
```

