```
//* Comparison Operators

//? 1 $eq: Matches values that are equal to the specified value.
// db.products.find({'price': {$eq:699}})

//? 2: $ne: Matches values that are not equal to the specified value.
// db.products.find({'price': {$ne:699}}).count()

// 3: $gt: Matches values that are greater than the specified value.
// db.products.find({'price': {$gt:699}})

// 4: $gte: Matches values that are greater than or equal to the specified value.
// db.products.find({'price': {$gte:699}})

// 5: $lt: Matches values that are less than the specified value.
// db.products.find({'price': {$lt:699}})

// 6: $lte: Matches values that are less than or equal to the specified value.
// Find products with price less than or equal to 699
// db.products.find({'price': {$lte:699}})

// $in: Matches values that are within the specified array.
// db.products.find({'price': 129, 'price':39})
// db.products.find({'price': {$in: [129,39]}})
//? Now here I will go with different collection
// db.category.find({ name: { $in: ["Travel & Luggage", "Home & Kitchen"] } });

//? $nin: Matches values that are not within the specified array.
// db.products.find({'price': {$nin: [249,129,39]}}).count()



// ****************************************
//* Index
// ****************************************
// lets query to see total count of products greater than 100:
// db.products.find({ price: { $gt: 100 } }).count();

//? We can use explain() method to understand it more better
//? Also we can add explain('executionStats') to understand more in depth

//? Find name= air fryer from the products collections
// db.products.explain('executionStats').find({'name':'Air Fryer'})
//! executionTimeMillis: 18,

//* Creating Indexes
//? Indexes can be created using the createIndex() method.
//? syntax: db.collectionName.createIndex({ fieldName: 1 });
//? In this case, 1 represents ascending order, and -1 would be descending order.

// db.products.createIndex({name:1})
//! executionTimeMillis: 8

//* Getting Indexes
// db.products.getIndexes();
//?Did you realize that _id is already there? _id is automatically added by mongodb
and it’s a default unique index.

//* Removing an index
```

```
// db.products.dropIndex({ name: 1 });

//* Creating a unique index
// db.users.createIndex({ email: 1 }, { unique: true });
//* When not to use indexes?
// 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.

// For example
// db.products.explain('executionStats').find({price: {$gt: 100}});
// The think it is returning almost 90% of data in output
// nReturned: 9216,

// let create the index for the Price
// db.products.createIndex({price:1})
// Now lets check the time its taking
// executionTimeMillis: 26, it is taking almost double time



//* Cursor Methods
// todo  We need to use () parenthesis in all the cursor methods
//? 1: count(): The count() method returns the count of documents that match the query.
// db.products.find({price: {$gt: 250}}).count()
//? 2: limit(): The limit() method restricts the number of documents returned by the query.
// db.products.find({price: {$gt: 250}}).limit(2)
//? 3: skip(): The skip() method skips a specified number of documents and returns the rest.
// db.products.find({price: {$gt: 250}}).limit(5).skip(1)
//? 4: sort(): The sort() method sorts the documents based on the specified field(s).
// db.products.find({price: {$gt: 1250}}).limit(3).sort({'price':1})



//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
//? 1:  Creating / Deleting Databases
// show dbs
// use thapaProducts
// db.dropDatabase()
//? 2:  Creating / Deleting Databases and Collections
// db.createCollection('test')
// show collections
// db.test.drop() // where the test is the collection name



//* Elements Operator
// 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.
//? 1: $exists: Matches documents that have a specific field, regardless of its value.
// db.products.find({ price: { $exists: true } }).count();
// Find documents with the "price" field present, and if it's present then check the value greater then 1200
```

```
// db.products.find({ price: { $exists: true },cls price: { $gt: 1250 } });
//? 2: $type: The $type operator filters documents based on the BSON data type of a field.
// Basically we need to search or find the fields based on types (BSON Type) for example
// db.products.find({ price: { $type: "string" } });
// result will be 0, bcz the price type is number
// db.products.find({ price: { $type: "number" } }).count()
// 1: Double
// 2: String
// 3: Object
// 4: Array
// 5: Binary data
// 6: Undefined
// 7: Object id
// 8: Boolean
// 9: Date
// 10: Null
// 11: Regular expression
// 12: JavaScript code
// 13: Symbol
// 14: JavaScript code with scope
// 17: 64-bit integer
// db.products.find({ price: { $type: "string" } });
// result will be 0, bcz the price type is number
// db.products.find({ price: { $type: "number" } }).count()
//? 3: $size: The $size operator matches documents where the size of an array field matches a specified value.
// db.comments.find({comments: {$size:2}})



//* Embedded Documents (Dealing with Arrays & Object)
//* Just use the dot notations, that’s it
// ?1:  Find posts with comments by a specific user (Array)
// db.comments.find({'comments.user': 'Alice'})
//? 2: Find the documents where the views in metadata field > 1200. (Object)
// db.comments.find({ "metadata.views": { $gt: 1200 } });
//? 3: we need to find out the document where the user in comments = Henry and also the in the metadata likes value > 50.
// db.comments.find({ 'comments.user':'Henry' , 'metadata.likes':{$gt: 50}  })
//? 4:we need to return an comments array which must have this two (user = alice & vinod) elements only in it.
//! We need to use $all operator. Here the order doesn’t’ matter.
// db.comments.find({ "comments.user": { $all: ["Alice", "Vinod"] } });
// db.comments.find({'comments.user': {$all: ['Alice','Vinod','Bob']}})
//? 5:  In Array for multiple querying we user $elemMatch operator.
//* Here is the syntax: {field: {$elemMatch: { {query1},{query2}..  }}}
// db.comments.find({ comments: {$elemMatch: {'user':'Alice' , 'text':'Awesome article!'  } }  })
//? by the wat you can write the same using simple way
// db.comments.find({ 'comments.user':'Alice' , 'comments.text':'Awesome article!' } )



//* Introduction to $expresion
// 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.
```

```
//? The syntax is {$expr: {operator: [field, value] } }
// One important thing to remember is the field should be prefix with $ sign.
// db.products.find({$expr: {$gt: ['$price',1340] }})
//! Find sales where (quantity * price) is greater than targetPrice
// db.sales.find({
//     $expr: {
//       $gt: [{ $multiply: ["$quantity", "$price"] }, "$targetPrice"],
//     },
//   });
// here both the values are fields only for comparison thats why $ sign is used


//* Logical Operators
//We have 4 logical operators with us. $and , $or, $nor and $not
// syntax: { operator:  [{condition1},{condition2},…] }
//? 1: $and: Performs a logical AND operation on an array of expressions, where all expressions must be true for the document to match.
//! Find products with price greater than 100 and name equal to "Diamond Ring"
// db.products.find({ $and: [ { 'price': { $gt: 10 } }, { 'name': 'Notebook Collection'} ] })
// db.products.find({'price': {$gt:10}, 'name':{$eq: 'Notebook Collection'}})
//* In MongoDB, when you provide multiple fields within a single query document, MongoDB treats them as an implicit AND operation.
//? 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.
// We can use logical operator only when we have the duplicate fields
// db.products.find({'price': 129, 'price':39})
// but we can write the same in $or operator
// db.products.find({ $or: [{ price: 129 }, { price: 39 }] });
//? 3: $not: Performs a logical NOT operation on the specified expression, inverting the result.
//? Find products with price not equal to 100
// db.products.find( {'price': {$not: {$eq: 100}} } )
//?4: $nor: Performs a logical NOR operation on an array of expressions, where none of the expressions must be true for the document to match
//* Find products with price not equal to 100 or name not equal to "Notebook Collection"
// db.products.find( {$nor: [ {'price': {$eq: 100}}, {'name':'Notebook Collection'} ]} )


//* MONGOIMPORT
// Now I will show How to import data from json file
//? mongoimport E:\\mongo\products.json -d shop -c products
//? mongoimport E:\\mongo\products.json -d shop -c products --jsonArray


//* Projection
// 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.
// 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.
// db.products.find({}, { name:1, price:1}).limit(2)
// 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.
// db.products.find({}, {_id:0, name:1, price:1}).limit(2)
//! We cannot include and exclude fields in the same query projection in MongoDB. It's either inclusion or exclusion, not both simultaneously.
// db.products.find({}, { _id: 0, name: 1, price: 1, price: 0 }).limit(2);

```

```
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:
```

```
//     {
//       _id: <expression>, // Group key
//       <field1>: { <accumulator1> : <expression1> },
//       ...
//     }
//  }

//
https://www.mongodb.com/docs/v6.0/reference/operator/aggregation/group/#considerations

db.products.aggregate([
  { $match: { price: { $gt: 900 } } },
  {
    $group: {
      _id: { sameCompany: "$company" },
      totalPrice: { $sum: "$price" },
    },
  },
]);

// let's use another accumulator operations
// $avg

// find the quantity = 5, group them with same quantity and find the average price

db.sales.aggregate([{ $match: { quantity: { $eq: 5 } } }]);
// both are same
db.sales.aggregate([
  { $match: { quantity: 5 } },
  {
    $group: {
      _id: { quan: "$quantity" },
      avgPrice: { $avg: "$price" },
    },
  },
]);

//* $sort

db.products.aggregate([
  { $match: { price: { $gt: 1200 } } },
  {
    $group: {
      _id: "$category",
      totalPrice: { $sum: "$price" },
    },
  },
  { $sort: { totalPrice: 1 } },
]);

// $sort is like .sort() but you can even sort the values that you added in group.
(Of course you can also sort before grouping or with any other values. But here you
can even sort in ascending or descending based on number of products it has.

db.products.aggregate([
  { $match: { price: { $gt: 1200 } } },
  {
    $group: {
      _id: "$category",
```

```
      totalPrice: { $sum: "$price" },
    },
  },
  { $sort: { totalPrice: -1 } },
]);

//* $project

db.products.aggregate([
  {
    $project: {
      _id: 0,
      price: 1,
      name: 1,
    },
  },
]);

// 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:

db.products.aggregate([
  { $match: { price: { $gt: 1000 } } },
  {
    $project: {
      _id: 0,
      name: 1,
      originalPrice: "$price",
      disPrice: { $multiply: ["$price", 0.8] },
    },
  },
]);

// again we can add the sort here too

db.products.aggregate([
  { $match: { price: { $gt: 1000 } } },
  {
    $project: {
      _id: 0,
      name: 1,
      originalPrice: "$price",
      disPrice: { $multiply: ["$price", 0.8] },
    },
  },
  { $sort: { disPrice: -1 } },
]);

//* $push and $unwind
//? Find documents with a price greater than 1200, then group them by price and create an array of colors for each group.

//* Before
//* if price = 1250 =>  colors: [ '#000000', '#cc6600', '#663300' ],
//* if price = 1250 =>  colors: [ '#fff000', '#ddddd', '#663300' ],

//? After,  I need a new document where
{
  price: 1250,
```

```
  allColors: ['#000000', '#cc6600', '#663300', '#fff000', '#ddddd', '#663300' ]
}

// code
db.products.aggregate([
  { $match: { price: { $gt: 1200 } } },
  {
    $group: {
      _id: { priceGroup: "$price" },
      colors: { $push: "$colors" },
    },
  },
]);

//* $unwind

db.products.aggregate([
  { $match: { price: { $gt: 1200 } } },
  { $unwind: "$colors" },
  {
    $group: {
      _id: { priceGroup: "$price" },
      colors: { $push: "$colors" },
    },
  },
]);

//?  Before
{
  _id: ObjectId("64c23601e32f4a51b19b9263"),
  name: 'Laptop Pro',
  company: '64c23350e32f4a51b19b9231',
  price: 1299,
  colors: [ '#333333', '#cccccc', '#00ff00' ],
  image: '/images/product-laptop.png',
  category: '64c2342de32f4a51b19b924e',
  isFeatured: true
},

//! $unwind: '$colors';
//? the $unwind stage deconstructs the "colors" array, creating multiple documents for each color within a product.

//?  After
{
  _id: ObjectId("64c23601e32f4a51b19b9263"),
  name: 'Laptop Pro',
  company: '64c23350e32f4a51b19b9231',
  price: 1299,
  colors: '#333333',
  image: '/images/product-laptop.png',
  category: '64c2342de32f4a51b19b924e',
  isFeatured: true
},

// {
//   _id: ObjectId("64c23601e32f4a51b19b9263"),
//   name: 'Laptop Pro',
//   company: '64c23350e32f4a51b19b9231',
//   price: 1299,
```

```
//   colors: '#cccccc',
//   image: '/images/product-laptop.png',
//   category: '64c2342de32f4a51b19b924e',
//   isFeatured: true
// },

// so now all the colors are in a string format, so $push will add them as an element in an array of colors

db.products.aggregate([
  { $match: { price: { $gt: 1200 } } },
  { $unwind: "$colors" },
  {
    $group: {
      _id: null,
      totalCount: { $sum: 1 },
    },
  },
]);

db.products.aggregate([
  { $match: { price: { $gt: 1200 } } },
  { $unwind: "$colors" },
  {
    $group: {
      _id: { priceGroup: "$price" },
      colors: { $push: "$colors" },
    },
  },
]);

//* $addToSet
// still there is a problem and that is we are also getting the duplicates values so to remove it we will use the $addToSet

db.products.aggregate([
  { $match: { price: { $gt: 1200 } } },
  { $unwind: "$colors" },
  {
    $group: {
      _id: { priceGroup: "$price" },
      colors: { $addToSet: "$colors" },
    },
  },
]);

//* $size
// What If we want to count the number of unique colors for each price group
db.products.aggregate([
  { $match: { price: { $gt: 1200 } } },
  { $unwind: "$colors" },
  {
    $group: {
      _id: { priceGroup: "$price" },
      colors: { $addToSet: "$colors" },
      colorLength: { $size: "$colors" },
    },
  },
]);

```

```
// 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.

db.products.aggregate([
  { $match: { price: { $gt: 1200 } } },
  { $unwind: "$colors" },
  {
    $group: {
      _id: { priceGroup: "$price" },
      allColors: { $addToSet: "$colors" },
    },
  },
  {
    $project: {
      _id: 1,
      allColors: 1,
      colorLength: { $size: "$allColors" },
    },
  },
  { $limit: 1 },
]);

//! 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

//* limit

db.products.aggregate([
  { $match: { price: { $gt: 1200 } } },
  { $unwind: "$colors" },
  {
    $group: {
      _id: { priceGroup: "$price" },
      allColors: { $addToSet: "$colors" },
    },
  },
  {
    $project: {
      _id: 1,
      allColors: 1,
      colorLength: { $size: "$allColors" },
    },
  },
  { $limit: 1 },
]);

//* skip

db.products.aggregate([
  { $match: { price: { $gt: 1200 } } },
  { $unwind: "$colors" },
  {
    $group: {
      _id: { priceGroup: "$price" },
      allColors: { $addToSet: "$colors" },
    },
  },
  {
    $project: {
```

```
      _id: 1,
      allColors: 1,
      colorLength: { $size: "$allColors" },
    },
  },
  { $skip: 1 },
]);

//* $filter
db.col.aggregate([
  {
    $project: {
      name: 1,
      values: {
        $filter: {
          input: "$values",
          as: "value",
          cond: { $gt: ["$$value", 30] },
        },
      },
    },
  },
]);

// Or Example 👎 //
db.produts.aggregate([
    {
      $group: {
        _id: "$company",
        totalProducts: { $sum: "$price" },
      },
    },
  ]);

  db.products.aggregate([
    {
      $match: {
        _id: "64c23350e32f4a51b19b9247",
      },
    },
  ]);


  price > 900
  company $group
  sum price

  db.produts.aggregate([
      {
          $match: {price: {$gt: 900}}
      },
      {
        $group: {
          _id: "$company",
          totalProducts: { $sum: "$price" },
        },
      },
    ]);

```

```
  //!   find the quantity = 5, group them with same quantity and find the average price

  db.sales.aggregate([
      { $match: {quantity:5} },
      {
          $group: {
              _id: '$quantity',
              priceTotal: {$sum: '$price'},
              pricrAvg: {$avg:'$price'}
          }
      }
  ])

  db.products.aggregate([
      { $match: { price: { $gt: 1200 } } },
      {
        $group: {
          _id: "$category",
          totalPrice: { $sum: "$price" },
        },
      },
      {
          $sort: {totalPrice: 1}
      }
    ]);


    db.products.aggregate([
      { $match: { price: { $gt: 1200 } } },
      {
          $project: {
              price:1,
              discountPrice: {$multiply: ['$price', 0.8]}
          }
      }
    ])

    db.products.aggregate([
      { $match: { price: { $gt: 1200 } } },
      {
          $group: {
              _id: '$price',
              allColors: { $push :'$colors'}
          }
      }
    ])

    price: 1999,
    colors: [ '#000000', '#cc6600', '#663300' ]

    price: 1999,
    colors: [ '#000000', '#cc6600', '#663300' ]


    price: 1999,
    colors: [
      [ '#000000', '#cc6600', '#663300' ],
      [ '#000000', '#cc6600', '#663300' ]
    ]
```

```
    ,
    price: 1999,
    colors: ['#000000', '#cc6600', '#663300']

    db.products.aggregate([
      { $unwind: '$colors' },
      { $match: { price: { $gt: 1200 } } },
      {
          $group: {
              _id: '$price',
              allColors: { $push :'$colors'}
          }
      }
    ])

    db.products.aggregate([
      { $unwind: '$colors' },
      { $match: { price: { $gt: 1200 } } },
      {
          $group: {
              _id: '$price',
              allColors: { $addToSet :'$colors'}
          }
      }
    ])

    db.products.aggregate([
      { $match: { price: { $gt: 1200 } } },
      { $unwind: "$colors" },
      {
        $group: {
          _id: { priceGroup: "$price" },
          colors: { $addToSet: "$colors" },
        },
      },
      {
          $project:{
              _id:1,
              colors:1,
              colorLength: { $size: "$colors" },
          }
      },
     {
      $limit: 1
     }
    ]);


    db.col.insertMany([
      {
        _id: "64c23350e32f4a51b19b9201",
        name: "Document 1",
        values: [10, 20, 30, 40, 50],
      },
      {
        _id: "64c23350e32f4a51b19b9202",
        name: "Document 2",
        values: [15, 25, 35, 45, 55],
      },
      {
```

```
        _id: "64c23350e32f4a51b19b9203",
        name: "Document 3",
        values: [5, 15, 25, 35, 45],
      },
      {
        _id: "64c23350e32f4a51b19b9204",
        name: "Document 4",
        values: [30, 40, 50, 60, 70],
      },
      {
        _id: "64c23350e32f4a51b19b9205",
        name: "Document 5",
        values: [25, 35, 45, 55, 65],
      },
    ]
  )


  db.col.aggregate([{
      $project:{
          name:1,
          thapaValue: {
              $filter:{
                  input: '$values',
                  as: 'val',
                  cond: {$gt: ['$$val', 30]}
              }
          }
      }
}])

// // CRUD Example  OPERATIONS IN MONGODB 👇 // //
// ** inserts Operation ** //

//? 3: crete (Inserting the documents in collection)
//!methods like insert() and save() are being deprecated in favor of more explicit methods like insertOne() and insertMany()

//? 1.a insertOne(): This method inserts a single document into the collection.
// db.product.insertOne({ name: "vinod", age: 29 });

//? 1.b insertMany(): This method inserts an array of documents into the collection.
//! IMP - Argument "docs" must be an array of documents (use array always)
//db.product.insertMany([{'name':'vinod', 'age':29}, {'name':'arjun', 'age':'30'}])

//? Important when to use quotes and when to not
//Special Characters: If your field name contains special characters, spaces, or starts with a numeric digit, using quotes becomes necessary.
// Field name with spaces
// db.grades.find({"course name": "Math"})
// Field name starting with a digit
// db.grades.find({"1st_place": true})

//Reserved Words: If your field name is a reserved keyword in MongoDB (e.g., $group, $sum, etc.), you need to use quotes to distinguish it from the reserved keyword
// we will see when we will see comparison operator

//? 1.c Ordered Inserts vs Unordered Inserts
//In MongoDB, "ordered" and "unordered" refer to the behavior of a bulk write 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.

//? it's a example of ordered Inserts after the 2nd execution it will stop
// db.product.insertMany([
//   { name: "vinod", age: 29 },
//   { _id: ObjectId("64cb3ea5be4cb31d576182a3"), name: "sujan" },
//   { name: "naran", age: "30" },
// ]);

//? Unordered Inserts
// db.product.insertMany(
//   [
//     { name: "vinod", age: 29 },
//     { _id: ObjectId("64cb3ea5be4cb31d576182a3"), name: "sujan" },
//     { name: "ram", age: "30" },
//   ],
//   { ordered: false }
// );
//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.

//? 3: Case Sensitivity in MongoDB
//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.
// db.Product.insertOne({name:"thapa",age:30})
// vs
// db.product.insertOne({name:"thapa",age:30})
// the output will be two collections 😯
// dbproduct> show collections
// product
// Product


// ** Read  Operations ** //

//* 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.
//? syntax => db.collection_name.find({key:value})
// db.product.find({ name: "vinod" });
// db.product.find({'age':29})

//* 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.
//? syntax => db.collection_name.findOne({key:value})
// db.product.findOne({ age: 29 });
// db.product.findOne({'name':'vinod'})

//* MONGOIMPORT
// Now I will show How to import data from json file
//? mongoimport E:\\mongo\products.json -d shop -c products

// mongoimport E:\\mongo\products.json -d shop -c products --jsonArray

//! Failed: error reading separator after document #1: bad JSON array format - found no opening bracket '[' in input source

```

```
// mongoimport E:\mongo\mongo_json\sales.json -d tshop -c sales --jsonArray

// mongoexport -c sales -d shop -o E:\mongo\sales1.json

// mongoexport --collection=sales --db=shop -out=E:\mongo\sales.json


// ** Delete Operations ** //
//? In MongoDB, the DELETE operations are used to remove documents from a collection. There are two main methods to perform deletion: deleteOne() and deleteMany().

//* Delete a Single Document:
//? syntax : db.collectionName.deleteOne({ _id: ObjectId("12345") });
// db.sales.deleteOne({ _id: 1 });

//* Delete Multiple Documents:
//? Syntax: db.collectionName.deleteMany({ field: "value" });
// db.sales.deleteMany({'price':55})


// ** Update Operations ** //
//? 1: Updating a Single Field:
//*  db.collectionName.updateOne(
//    { _id: ObjectId("12345") },
//   { $set: { fieldName: "new value" } }
//  );

//? Update the price value = 45 in a products collections, where the _id = ObjectId("64c2363be32f4a51b19b9271")

//? Update the isFeatures value = true in a products collections, where the name = Designer Handbag

//* UpdateMany
//? Update all the isFeatures value = true in a products collections, where the price = 120

//* Updating multiple fields in a document
// db.collectionName.updateOne(
//   { _id: ObjectId("12345") },
//   {
//     $set: {
//       field1: "new value 1",
//       field2: "new value 2",
//     },
//   }
// );

//? Update the price = 154 and isFeatures = false fields from the products collections where the name =  Unbranded Frozen Chicken.

//* Renaming a field in a document.
// syntax:  db.collectionName.updateOne(
//     { _id: ObjectId("12345") },
//     { $rename: { oldFieldName: "newFieldName" } }
//   );
//? Rename the products collection isFeatured field to isFeature, where the price = 123

```

```
//* Adding a new field in a document
// db.collectionName.updateOne(
//     { _id: ObjectId("12345") },
//     { $set: { newField: "new value" } }
//   );

//* Removing or Deleting the Field in a document
// To remove a field from documents in MongoDB, you can use the $unset update operator.
// db.collectionName.updateOne(
//     { _id: ObjectId("12345") },
//     { $unset: { fieldName: 1 } }
//   );

//* Update Embedded Documents
//? How do you add a new element to an array using the $push operator?
// db.collectionName.updateOne(
//     { _id: ObjectId("12345") },
//     { $push: { arrayField: "new element" } }
//   );

//? Popping from an Array: Removing the last element from an array in a document.
// Syntax:  db.collectionName.updateOne(
//   { _id: ObjectId("12345") },
//   { $pop: { arrayField: 1 } }
// );

//? Updating a field within an embedded document.

//? Update the text value within an comments array = "Awesome article!", where the id=7 & username=alice.

// Consider this part of the query: 'comments.$.text': 'Awesome Thapa!'

// comments is the name of the array field.
// $ is the positional operator, and it refers to the index of the array element that matches the query condition.
// text is the field within the specific comment element that you want to update.


// **** Monodb Opration Relation **** //
// const { MongoClient } = require("mongodb");
// const uri = "mongodb://127.0.0.1";
// const client = new MongoClient(uri);

// const data1 = {
//   name: "Designer Handbag1",
//   company: "64c23350e32f4a51b19b923a",
//   price: 3466,
//   colors: ["#000000", "#cc6600", "#663300"],
//   image: "/images/product-handbag.png",
//   category: "64c2342de32f4a51b19b9250",
//   isFeatured: true,
// };

// const main = async () => {
//   await client.connect();
//   const db = client.db("shop");
//   const collection = db.collection("products");

```

```
//   await collection.insertOne(data1);

//   const data = await collection.find({ price: { $eq: 3466 } }).toArray();
//   console.log(data);
//   return "done";
// };

// main()
//   .then(console.log())
//   .catch((e) => console.log(e))
//   .finally(() => client.close());



// **** Mongoose to Relation Operation **** //
const mongoose = require("mongoose");

// const uri = "mongodb://127.0.0.1/shop";

const uri =
  "mongodb+srv://vbthapa55:qwerty123@cluster0.kziyfmu.mongodb.net/shop?retryWrites=true&w=majority";

mongoose.connect(uri);

// we need to create a schema
const productSchema = new mongoose.Schema({
  name: String,
  company: String,
  price: Number,
  colors: [String],
  image: String,
  category: String,
  isfeatured: Boolean,
});

// we need to now create an model
const Product = new mongoose.model("Product", productSchema);


// 💔 Thank You So Much For Choosing My Video 💔

// Hi everyone,

// 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.

// 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!
// Here is the link: https://www.youtube.com/thapatechnical

// With gratitude,
// Thapa Technical



//? 2nd step while inserting the data
const data1 = {
```

```
  name: "Designer Handbag2",
  company: "64c23350e32f4a51b19b923a",
  price: 2466,
  colors: ["#000000", "#cc6600", "#663300"],
  image: "/images/product-handbag.png",
  category: "64c2342de32f4a51b19b9250",
  isFeatured: true,
};

const main = async () => {
  try {
    //? 2: insert documents
    // await Product.insertMany(data1);
    // const data = await Product.find({ price: { $eq: 2466 } });
    // console.log(data);

    //? 3 update query
    // await Product.findOneAndUpdate(
    //   { name: "Designer Handbag2", price: 2466 },
    //   { $set: { price: 2499 } }
    // );

    //? 3 Delete query

    await Product.findOneAndDelete({ name: "Designer Handbag2", price: 2499 });
    const data = await Product.find({
      name: "Designer Handbag2",
      price: 2499,
    });

    console.log(data);
  } catch (error) {
    console.log(error);
  } finally {
    mongoose.connection.close();
  }
};

main();

db.Students.insertMany(
  [{
    name: "Binamra",
    age: 20,
  },
  {
    name: "Thapa",
    age: 21,
  }]
);

 {
    "name": "Binamra",
    "age": 20,
  }
  {
    "name": "Thapa",
    "age": 21,
  }
```