The Magical Marvels of MongoDB

About
MongoDB acts as the database that stores data for your application. Explore the basics and learn to store data in a document-oriented database.

Conjuring MongoDB (Level 1)

First things first, we need to define what MongoDB is. MondoDB is an open-source NoSQL database. A NoSQL database is a database that usually aren't relational and don't have a query language like SQL. MongoDB is document-oriented database, which means it is suitable for storing/managing collections of documents like text document, email messages, XML document, etc. Essentially, MongoDB is a good choice if you have a lot of unstructured data.

Lets compare MongoDB to an SQL database. In SQL database, your data is put into rows, which is put into a table, which goes into the database. That way you're able to call for data from a specific table and a specific row to get a collection of data that matches your query. Into MongoDB, your data is put in a document, which is put into a collection, which goes into the database.

<table>
<tr><th>Relational</th><th>Document-oriented</th><th>Notes</th></tr>
<tr>
<td>Apples Table
<table>
<tr><th>apple_id</th><th>name</th><th>price</th><th>vendor_id</th></tr>
<tr><td>1</td><td>"Granny Smith"</td><td>0.99</td><td>1</td></tr>
<tr><td>2</td><td>"Red Delicious"</td><td>1.29</td><td>2</td></tr>
<tr><td>3</td><td>"Honey Crisp"</td><td>1.09</td><td>3</td></tr>
</table>
Vendors Table
<table>
<tr><th>vendor_id</th><th>name</th></tr>
<tr><td>1</td><td>"Farmer Fred"</td></tr>
<tr><td>2</td><td>"Farmer Ted"</td></tr>
<tr><td>3</td><td>"Farmer Sam"</td></tr>
</table>
</td>
<td>Apples Collection
Document
Name: "Granny Smith"
Price: 0.99
Vendor: "Farmer Fred"

Document
Name: "Red Delicious"
Price: 1.29

Document
Name: "Honey Crisp"
Vendor: "Farmer Sam"</td>
<td>You can see how in a Relational Database, you sort your information into tables with rows that contain your data. Note that your data needs to be able to fit in the tables to exist in the database.

But in a Document-Oriented Database, you put the information in its own document which is added to a collection. Note that your data doesn't have to have exact fields to be added to the database.</td>
</tr>
</table>

As you can see in the example above, within a Collection, your documents can have different fields in them. You can have a Name, Price, Vendor, Notes, Dangers, or any fields you want. Being able to have different fields is known as "dynamic schema". This is MongoDB's advantage, because you can have a lot of data that is related in some way, but not exactly the same.

Starting the Shell `1 > mongo`	You can access the MongoDB through the shell. Once started, your mongo commands will start after the ">".

Documents in a MongoDB are JSON-like Objects. You will have a field, this this case "name" and "vendor", and a value that corresponds to a field, in this case "Fuji" and "Farmer Fred". You can have as many field-value pairs as you want, just remember to separate them with a comma.	`1 {`
	`2  "name": "Fuji",`
	`3  "vendor": "Farmer Fred"`
	`4 }`

Shell: Reviews `1 > use <name>`	This shell command will specify which database we want to use. In the event you have multiple databases, you'll need a way to switch between them. If the database doesn't already exist, Mongo will make it for you.
Shell: Current Database `1 > db`	This shell command will tell you which database you are currently in.
Shell: Help `1 > help`	This shell command will bring up a list of MongoDB shell commands you can use.

Shell: Insert Document into a Collection
`> db.<collection>.insert(`
`  {`
`  <JSON data>`
`  }`
`)`
	This shell command will insert a Document into a collection. Note that when you interact with the database, you specify bd.<collection> to tell the database you want to interact with a specific collection. Like the use command, if you haven't already created your collection, Mongo will create it for you during this command. After entering this command, Mondo will return a "WriteResult Object", whose purpose is to tell us if the write was successful or not.

Shell: Find all Documents `> db.<collection>.find()`	This shell command will retrieve all documents in a specified collection. Do not be surprised when your documents suddenly have a "_id" field, as Mongo needs all Documents to have a unique identifier. If you don't specify an _id, Mongo will generate one for you.

Shell: Perform a Query
`> db.<collection>.find(`
`  {<field>: <value>}`
`  )`
	This a way to specify you want to find documents in a collection that contain a certain key-value pair expressed in JSON form. Remember that queries are case sensitive.
	Ex. `db.<collection>.find({"name": "example"})`

It was mentioned earlier that documents in MongoDB are stored as JSON objects, but its not exactly accurate. Documents in MongoDB are actually stored as BSON objects. Like in JSON you can store strings, numbers, booleans, arrays, objects, and null, in addition, BSON can store ObjectID and ISODate.

MongoDB supports embedded objects, such as the "ratings" object seen to the right. Embedded objects no not need a unique _id. To query for the "flavor" value in the object to you write, you need to use dot notation, as seen below.

```
db.apples.find({"ratings.flavor": 7})
```

Since "flavor" is a value of the object that is an object for "ratings", you can use the value.value notation to specify that's what you want.

```
{
  "name": "Fuji",
  "vendor": "Farmer Fred",
  "price": 2.99,
  "score": 78
  "tryDate": new Date(2014, 09, 27),
  "helpers": ["Bob", "Jim", "Sam"],
  "ratings": {"flavor": 7, "color": 13}
}
```

It may surprise you, but MongoDB doesn't enforce many validations on the documents you can put into it. There are only a few things you actually need to do: have unique _id field, have no syntax errors, and the document is less than 16 mb. Mongo need to be able to identify one and only one document with any given _id, which is why you can't have duplicates, it wouldn't know what to do. You need to be syntactically correct so the JavaScript works when its processed. And you need to have you documents be small, so an excessive amount of RAM isn't required.

Mystical Modifications (Level 2)

There is going to a time, where for one reason of another, you will need to remove a document from your database. Perhaps the information is no longer valid, or maybe the document is causing problems for the rest of your database.

Shell: Remove a Document
`> db.<collection>.remove(`
`  {<JSON-query>}`
`)`
	This shell command will go into the specified collection, and remove documents that match the given query. Much like the insert method, you will receive a WriteResult object that will tell you if the remove was successful, and how many were if it was.

Shell: Update a Document
`> db.<collection>.update(`
`  {<JSON-query>},`
`  {"$set": {<field>: <value>}}`
`)`
	This shell command will query a collection, and will set the documents that match the query with the given update parameter. Note that only the first matching document will be updated.
	Ex.
	`> db.apples.update(`
	`  {"name": "Fuji"}, {"$set": {"price": 2.99}}`
	`)`
	Note that when the WriteResult is returned for an update, it will tell you the number of documents that matched your query, the number of documents that were created, and the number of documents that were modified.

Shell: Updating Without an Order

```
> db.apples.update(
    {"name": "Fuji"},
    {"price": 2.99}
  )
```

Before Update

Name: "Fuji"
Vendor: "Bob"
Price: 4.99

After Update

Price: 2.99

If you do not specify the "set" operator when doing an update, then everything but the _id field will be replaced with the update parameter.

This can be useful when you are importing or overwriting data though.

Shell: Updating Multiple Documents

```
> db.<collection>.update(
    {<JSON-query>},
    {<field>: <value>},
    {"multi": true}
  )
```

This shell command is just the update command, but the third parameter passed, the option parameter. In this case, we are telling the function to update more than the first match. When the WriteResult is returned, you will see that the number of matches will be the same as the number of changes.

Lets continue with the Apples Example into the next part. You want to be able to see how many view counts your apples have gotten online, to see what's most popular.

Shell: Update a Document's Count

```
> db.<collection>.update(
    {<JSON-query>},
    {"$inc": {<field>: <value>}}
  )
```

This shell command will take in a JSON query to match documents, but is passed the inc operator. The inc operator will increment the passed parameter in either the positive or negative direction.
Ex. `{"$inc": {"count": 1}}` Would increase the count by 1
If the <update-parameter> doesn't already exist, it will be created in the update.

Shell: Find or Create a Document

```
> db.<collection>.update(
    {<JSON-query>},
    {"$inc": {<field>: <value>}},
    {"upsert": true}
  )
```

This shell command takes the option parameter to "upsert" if a match doesn't already exist. Should a document be created, it will take in the <JSON-query> and the <update-parameter>, that will be added to the newly created document. When the WriteResult object is returned, it will say that 1 object was upserted during the update.

While working in your database, there could come a time when the fields in your documents need to be altered. You might need to start tracking something new, or what you were tracking has been reclassified and needs to be renamed, or perhaps the field you were watching is no longer needed so it should be removed.

Shell: Remove a Field

```
> db.<collection>.update(
    {},
    {"$unset": {<field>: ""}},
    {"multi": true}
  )
```

This shell command will remove the <field> from all documents in the collection. We are specifying the JSON-query as {} to ensure that every document in the collection is a match for the update. Like other updates, it is the "$unset" operator that tells the update to remove the field. Note that the value for the field we pass in doesn't matter, it won't impact the operation.

Shell: Rename a Field

```
> db.<collection>.update(
    {},
    {"$rename": {<field>: <rename>}},
    {"multi": true}
  )
```

This shell command will rename a field name for documents in a collection. Again, we pass in an empty JSON-query to ensure that every document in the collection is a match. We use the "$rename" operator to set the current field name <field> to the name we want it to be <rename>. And the "multi" option is set to true to hit multiple documents.

Lets say we want to update the values in an array in our document. We can't use the "$set" operator as we have been so far, since `{"$set": {<field>: <value>}}` will set the field (our array) to the value we pass into it.

Shell: Regular Update on Array

```
{
  "_id": ObjectId(...),
  "name": "Fuji",
  ...
    "helpers": ["Jim", "Bob"]
}
```

```
> db.apples.update(
    {"name": "Fuji"},
    {"$set": {"helpers": "Tim"}}
  )
```

Using the update here will result in the document resulting in what's shown to the right.

```
{
  "_id": ObjectId(...),
  "name": "Fuji",
  ...
  "helpers": "Tim"
}
```

Shell: Indexed Update on Array

Since we are dealing with an array, we'll need to tell the operator which element we want to update. In our example above, "Jim" has index 0 while "Bob" has

```
> db.apples.update(
    {"name": "Fuji"},
    {"$set": {"helpers.0": "Tim"}}
  )
```

Using an indexed update will result in the document that's shown to the right.

```
{
  "_id": ObjectId(...),
  "name": "Fuji",
  ...
  "helpers": ["Tim", "Bob"]
}
```

index 1.

But what if we wanted to update the same value in arrays in multiple document, but the value was in different indexes.

Shell: Placeholder Operator

```
> db.<collection>.update(
    {<JSON-query>},
    {"$set": {<field>.$: <value>}},
    {"multi": true}
  )
```

This shell command will update the value, without needing a specified index. This is especially useful for when you have arrays with the same value in them, but at different indexes. The "$" acts as a placeholder, so the command simply looks for the JSON-query value we want to change. Note this will only update the first instance of the value in the array.

Shell: Updating Embedded Value

```
> db.<collection>.update(
    {<JSON-query>},
    {"$set": {<field>.<embedded>: <value>}}
)
```

This shell command will update the embedded value in a Document.

```
"ratings": {
             "flavor": 5,
             "color": 7
           }
```

Similar to the index used earlier, we need to specify we want to update the embedded value in a field, which we do like so:

```
{"$set": {"ratings.color": 9}}
```

Shell: Useful Update Operators

```
> db.<collection>.update(
    {<JSON-query>},
    {<op>: {<field>: <value>}}
  )
```

These are a few useful shell commands to know for updating.
"$max": Will update if new value is greater than current. Will insert if empty.
"$min": Will update if new value is less than current. Will insert if empty.
"$mul": Multiplies current field value by specified value. Will insert 0 if empty.

You can learn about more Update Operators here: https://docs.mongodb.com/manual/reference/operator/update/

We're going to move forward with working with Arrays.

Shell: Pop Operator

```
> db.<collection>.update(
    {<JSON-query>},
    {"$pop": {<field>: <1 or -1>}}
  )
```

This shell command will remove either the first or last element from an array. You specify this by passing it 1 for the last element, or -1 for the first element. This can use useful for removing at either end of the array without needing to know the indexes. Note that the "$pop" operator doesn't return the value that is popped, it merely modifies the array.

Shell: Push Operator

```
> db.<collection>.update(
    {<JSON-query>},
    {"$push": {<field>: <value>}}
  )
```

This shell command will add the passed value to the end of an array. This is useful of easily adding new elements to an array with only knowing the <field> name.

Shell: addToSet Operator

```
> db.<collection>.update(
    {<JSON-query>}:
    {"$addToSet": {<field>: <value>}}
  )
```

This shell command will add the passed value <value> to the end of an array, unless it is already present in the array. This is useful for making an array with non-repeating items.

Shell: Pull Operator

```
> db.<collection>.update(
    {<JSON-query>},
    {"$pull": {<field>: <value>}}
  )
```

This shell command will remove values from an array. Note that all instances of the value <value> present in the array will be removed.

Materializing Potions (Level 3)

Earlier we discussed how to query your database to find documents that matched a criteria. But there could be hundreds of returns that match our query that we need to go through. Fortunately, we can add a filter to further specify what we want our query to return.

Shell: Query with Multiple Criteria

```
> db.<collection>.find(
    {<field>: <value>, ...}
  )
```

This shell command will take in as many comma separated queries as you pass into it, and return documents that match all the queries.

Shell: Comparison Query Operators
`1 > db.<collection>.find(`
`2 {<field>:`
`3 {<op>: <value>, ...}`
`4 }`
`5 )`
	This shell command will use comparisons query operators to match document whose <field> compares to the passed value <value>.
	You can use "$gt" (greater than), "$lt" (less than), "$gte" (greater than or equal to), "$lte" (less than or equal to), "$ne" (not equal to) in place of <op>.
	Note that you are able to pass in more than one comparison operator, as long as it is syntactically correct and is comma separated.

Shell: Range Queries on an Array
`1 > db.<collection>.find(`
`2 {<field>: {"$elemMatch":`
`3 {<op>: <value>, ...}}`
`4 }`
`5 )`
	If we want to be able to search the "weights" field for the document to the left, we're going to need to introduce the "$elemMatch" operator. This will return documents where at least 1 element in the array matches all the criteria. Again, we ca pass multiple operators ($ge, $le, etc.) as long as they are comma separated.

So far, we've been using the find function to return the full documents that match the passed in query. That can be a lot of information to look at if we only want to look at specific fields though. Luckily, MongoDB has Projections, a way to return only the fields we want to see.

Shell: Query Including Fields
`1 > db.<collection>.find(`
`2 {<JSON-query>},`
`3 {<field>: true, ...}`
`4 )`
	This shell command is a continuation of the find command. Note that the second parameter in the find command contains a list of fields we want returned when we run out query. Note that you can pass in multiple fields to return, comma separated, and that _id is automatically the only field set to true by default, meaning you'll need to specify the fields you want manually.

Shell: Query Excluding Fields
`1 > db.<collection>.find(`
`2 {<JSON-query>},`
`3 {<field>: false, ...}`
`4 )`
	This shell command will return all fields, except those you manually set to false. This is useful for wanting to see all but one field, perhaps a long list you need to scroll through. Note that when excluding, all fields except those set to false are defaulted to true. Note that removing the _id is common when preparing data reports for non-developers.

Note that you cannot mix setting fields to true and false in the same query, this will result in an error. The only exception to this is specifying true for fields you want, and setting false for the "_id" field.

When you run the find function, what is returned is a cursor object. And in that cursor object, only the first 20 documents are initially seen. You will need to tell Mongo that you want to see more.

Shell: Iterate the Cursor `1 > it`	This shell command will display, in batches of 20, the results of a find query. Initially only 20 will be shown, so you will be prompted to see the next batch until there are no documents left.
Shell: Cursor Count `1 > db.<collection>.find(...).count()`	Since the cursor is an object, we are able to chain methods onto the end of it. As shown to the left, is the count method chained to a find query, which will return the count of the cursor object returned by the find method.
Shell: Cursor Sort `1 > db.<collection>.find(...).sort({<field>: -1 or 1})`	This shell command will sort the results of the find method. You specify -1 for descending order, and 1 for ascending order.

If you have a large database, you can very easily return hundreds of results from a filtered query. Being able to limit the number of documents that are shown after a query is called pagination.

Shell: Pagination Limit `1 > db.<collection>.find(...).limit(<limit>)`	This shell command will limit the number of documents that are shown when you run a query and are returned results. This can be seen as specifying how many items will appear on a page at most.
Shell: Pagination Skip `1 > db.<collection>.find(...).skip(<skip>)`	This shell command will skip over, and not display, a number of documents that are shown when you run a query. This is useful for not showing the same item of multiple pages, but skipping over it once shown.

We're going to look at an example that returns 9 documents from the find method. Below is visualization of what it would look like to chain limits and skips to the find method. The area boxed in is the representation of the documents that would be included after the limits and skips are applied.

`db.<collection>.find(...).limit(3)`	D1 D2 D3 D4 D5 D6 D7 D8 D9
`db.<collection>.find(...).skip(3).limit(3)`	D1 D2 D3 D4 D5 D6 D7 D8 D9
`db.<collection>.find(...).skip(6).limit(3)`	D1 D2 D3 D4 D5 D6 D7 D8 D9

Morphing Models (Level 4)

Through the first three levels, we have discussed how to create, add to, and modify our database. We then talked about how to view and filter through it, an even chain methods when doing so. In this section we will be discussing how to best model our date in the database.

```
{
  "name": "Fuji",
  "vendor": "Farmer Fred",
  "price": 2.99,
  "score": 78
  "tryDate": new Date(2014, 09, 27),
  "helpers": ["Bob", "Jim", "Sam"],
  "ratings": {"flavor": 7, "color": 13}
}
```

Shown to the left is a basic apples document that we have been using. You can note that the vendor, Farmer Fred, always has the same helpers work him to harvest these apples. This will result in a lot of duplicated data, which can be difficult to keep track of. This is known as Data Integrity.
There exists the possibility of making changes that do not persist throughout all the documents. Meaning that there could be inconsistencies in your data, rendering it problematic. We need a way to ensure that this doesn't happen.

One way to keep handle this is to reference information from another document.

Document in Apples Collection

```
{
  "_id": ObjectId(...),
  "name": "Fuji",
 "vendor_id": "FredCo",
  "price": 2.99,
  "score": 78
  "tryDate": new Date(2014, 09, 27),
  "ratings": {"flavor": 7, "color": 13}
}
```

Document in Vendor Collection

```
{
  "_id": "FredCo",
  "helpers": ["Bob", "Jim"]
  "phone": 1-234-567-8900
}
```

What you're seeing here is actually something from relational databases. You make a reference to another data entry, so that if your vendor (in this case) changes, the changes will follow through to every apple that references that vendor. It's a way to reduce the possibility of errors and inconsistencies.

To insert a referenced document into your database, you need to use the insert method. Remember to insert the (in this case) vendor into the vendor collection first, since it is what's being referenced. Then you would insert the apple into the apples collection that makes a reference to a document with a unique _id that already exists.

To query a referenced document in your database, you need to use the find method. You'll first find in the apples collection the apple you want, which will contain a reference to a vendor. You'll then need do another find on the newly acquired _id in the vendor collection to see the data related to the vendor.

The benefit to using embedded documents, is that when you query for the field using inclusion/exclusion we get it all. We get all of the information for the array or object, and when we add to it, it will appear with the same call we used earlier.

MongoDB has atomicity, which means if an error occurs then nothing will change. This is especially useful when doing an update. If you tried to make an update, but ran into an error, you wouldn't get a partial update that stopped when the error occurred. The document would remain unchanged, ready for you to come back and try again. On the other hand, if a WriteResult says an update occurred, then it successfully occurred in full without error.

In MongoDB, referenced documents exist independently. This means that even if our apple documents references a vendor document, we could still use the vendor document without needing to use the apple document. They aren't tied to each other, they can both be used without the other if needed. This is useful, since you can update your information in one place, which will help to reduce the possibility of inconsistencies in your database.

Something to keep in mind, is MongoDB does not support multi-document writes. Since Mongo simply keeps a record of field value

pairs, it doesn't know if a failed write happened to a document that is being referenced. It will still try to look for a document that might not exist because it wasn't added/updated properly, which could be a problem later on.

When you start making a database, you'll need to decide if its best to use embedding or references for relationships between documents. There are benefits and drawbacks to both, so you need to know how your data interacts to make the best choice.

Embedding	Referencing
• Perform single queries to get all our data. • Embedded documents are accessed through its parent, meaning it is not independent. • Atomic writes are supported, meaning that changes to a document will ensure the change carries over.	• Need 2+ queries to get all our data. • Referenced documents are independent, meaning you do not need one to access the other. • Multi-document writes are not supported, meaning that a successful and unsuccessful changes will cause issued.
• Useful when data is always used together, sometimes used together, or rarely used together. Meaning that if some data is needed in another data, its better to embed it.	• Useful when data is sometimes used together or rarely used together. Meaning that if the data sometimes is used together, but can be used without the other, its better to reference.
• Useful when there are less than 100 or more than a few hundred documents. This is because fully loading an embedded document takes time and resources.	• Useful when there are more than a few hundred or thousands of documents. Meaning that you only need load a reference to where the related data is, saving you time and resources.
• Useful when your data rarely or occasionally changed. This is because you need to make a full change to the entire document and its embedded information, which could be difficult to manage.	• Useful when your data occasionally or constantly changes. This is because you only need to make a change in one place, and it will appear in every place its referenced correctly, making it easy to prevent duplications.

Here are a few summaries to keep in mind when decide to reference or embed documents.

- Generally, embedding is the best starting point, since you'll want all of your information in one easy-to-access place.
- You should reference document when you need to access the documents independently.
- You should consider referencing your document when you have large, constantly changing data.
- Another thing to keep in mind, is if you need complex references in you document, you might want to consider using a relational database instead of a document oriented database.

Aggregation Apparitions (Level 5)

Lets say we're working with our database of vendors and apples. We want to know how many number of apples that each vendor sells. We could use a find query with field inclusion and pass the result into an Excel spreadsheet to get out answer. But it turns out MongoDB already has the tools needed for this, The Aggregation Framework, a way to perform complex calculations.

Shell: Group Aggregation

```
1 > db.<collection>.aggregate(
2     [{"$group": {<key>: $<groupBy>}}]
3   )
```

This shell command will group documents based on a passed in group key <key> and a field to group by <groupBy-field>. You tell the command the field <key> you want to use from your collection, and the unique group by field (that needs a "$") to be returned.

Apples Collection

```
1 {"_id": ObjectId(...), "name": "Fuji", "vendor_id": "Tim"},
2 {"_id": ObjectId(...), "name": "Gala", "vendor_id": "Sam"},
3 {"_id": ObjectId(...), "name": "Red", "vendor_id": "Tim"}
```

To the right you can see a group aggregation being applied to the collection above. We tell the shell to return a list of unique vendors, grouped together by the "vendor_id" field that exists in the documents.

Aggregation

```
1 > db.apples.aggregate(
2     [{"$group": {
3         "_id": "$vendor_id"}}]
4   )
```

Results

```
1 {"_id": "Tim"}, {"_id": "Sam"}
```

Shell: Sum Accumulators

```
1 > db.<collection>.aggregate([
2     {"$group": {<key>: $<groupBy>,
3      "total": {"$sum": 1}}}
4   ])
```

This shell command adds on an accumulator to the group aggregate function. The accumulator will increment by 1 for each matching document that passes through it. Note that you can have multiple comma separated accumulators.

Apples Collection

```
1 {"_id": ObjectId(...), "name": "Fuji", "vendor_id": "Tim"},
```

Aggregation

```
1 > db.apples.aggregate([
```

```
{"_id": ObjectId(...), "name": "Gala", "vendor_id": "Sam"},
{"_id": ObjectId(...), "name": "Red", "vendor_id": "Tim"}
```

This is an example of an accumulator added onto the last shell command. It will keep track of the number of documents that have each unique vendor in them.

```
    {"$group": {"_id": "$vendor_id",
     "total": {"$sum": 1}}}
])
```

Results

```
{"_id": "Tim", "total": 2},
{"_id": "Sam", "total": 1}
```

Over the course of this course, we've seen that the "$" has appeared in a few places. Its important to remember that when it appears in a field ($group, $sum, $pop, $push, $pull, etc.) it represents an operator that will perform a task. If however it appears in a value ($vendor_id) it will represent field paths that point to the value.

Shell: Other Accumulators

```
> db.<collection>.aggregate([
    {"$group": {<key>: $<groupBy>,
     <acc>: {"$<acc>": $<field>}}}
])
```

This shell command will instead of using the $sum accumulator, use a variety of other accumulators <acc>.
$acc - will calculate the average of a field of matching documents
$max - will return the max of a field of matching documents
$min - will return the min of a field of matching documents

The aggregate method that we have been using acts in a way like a pipeline. In that we can pass data through stages upon stages in order to change it along the way.

Shell: Aggregate Pipeline

```
> db.<collection>.aggregate([<stage>, <stage>, ...])
```

This is a representation of the aggregation pipeline, and the stages that the data passes through on its way to completion. We have already looked a bit at this, the $group is actually a stage in the aggregation pipeline.

Shell: Aggregation $match Stage

```
> db.<collection>.aggregate([
    "$match": {<field>: <value>, ...}
])
```

This stage in the aggregation pipeline works like a normal query. In that only documents that meet the specified field value pairs will continue on. Note that its good practice to use the $match stage earlier than later in you pipeline. Note that you can use $match multiple time.

Shell: Aggregation $sort Stage

```
> db.<collection>.aggregate([
    {"$sort": {<key>: -1 or 1}}
])
```

This stage in the aggregation pipeline simply sorts the field <key> that you pass into it. You pass in -1 to sort in descending order, and pass in 1 to sort in ascending order.

Shell: Aggregation $limit Stage

```
> db.<collection>.aggregate([
    {"$limit": <limit>}
])
```

This stage in the aggregation pipeline simply limits the number of documents <limit> to move on.

Shell: Aggregation $project Stage

```
> db.<collection>.aggregate([
    {"$project": {<field>: true or false, ...}}
])
```

This stage in the aggregation pipeline projects the fields we either want to keep in or remove before going onto the next stage. Note that you need to follow projection rules in that stage. Like $match, its good practice to have the $project stage earlier than later.