MongoDB Notes - Continuation

https://www.codecademy.com/learn/learn-mongodb

What is MongoDB?

MongoDB - No SQL database, uses JavaScript syntax. No SQL database is non-relation database.

MongoDB allows users to store data using the **document model**. The document model is a term used to describe a database that primarily stores data in documents and collections. The data stored inside documents is typically stored in hierarchical structures like JSON, BSON, and YAML.

What does it mean that a database follows the document model?

The document model is a database structure (for NoSQL databases) where the data are stored as documents within collections. Each entry in the database takes the form of a single document that stores certain information. The fields any one document stores is not dependent on any other document. Two documents in the same collection might store completely different fields. Typically the data is stored in JSON-like structures.

What is MongoDB Atlas?

MongoDB Atlas is MongoDB's multi-cloud database service.

The usage of Database:

- Databases are systems that store, modify, and access structured collections of information electronically.
- Database Management Systems (DBMS) allow developers to communicate via code or a graphical user interface with a database.
- Databases can store a wide range of data types, including text, numbers, dates and times, and files of various types.

Difference between Relational & Non-Relational Databases:

Relational databases structure data into tables with columns and rows and allow developers to pre-define relationships between these groups of data. These databases are highly accurate and flexible but can have slower performance for complex queries and are costly to maintain. (SQL)

Alternatively, non-relational databases refer to any database that does not follow the relational model of organized data into tables with rows and columns. The loosely defined structure of these databases makes them highly scalable, and they are cheaper to maintain, but accuracy can be a challenge to maintain, and queries tend to be less flexible. (No SQL)

JSON advantages and disadvantages

JSON's usefulness as a format for data comes primarily from the ease with which users can view and modify it. Data is stored in an easily editable format that is totally comprehensible to humans as well as our computers. It can be changed fairly easily with little fuss. However, it has several disadvantages, including slow performance, poor storage efficiency, and limited data types.

BSON advantages and disadvantages

Binary JavaScript Object Notation - BSON, is the format that MongoDB uses to store data. BSON is different than JSON in three fundamental ways:

1. BSON is not human-readable.
2. BSON is far more efficient storage-wise.
3. BSON supports a number of data formats that JSON does not - like dates.

something like this:

“\x00\x00\x00\x02name\x00\a\x00\x00\x00Rodney\x00\x02occupation\x00\r\x00\x00\x00photographer\x00\x10year_of_experience\x(

NoSQL (Not only SQL)

https://www.codecademy.com/courses/learn-mongodb/articles/introduction-to-nosql

CRUD 1

Finding Documents

https://www.codecademy.com/courses/learn-mongodb/lessons/crud-i-finding-documents

Browsing and Selecting Collections

MongoDB is one of the easiest databases to get started with! MongoDB can easily be run in a terminal using the MongoDB Shell (`mongosh` for short). Throughout this course, we will be providing you with your very own `mongosh` shell via a terminal. Now - before we can get into making fancy queries on our data, one of the first things we will have to do is navigate around our database instances. MongoDB allows us to store multiple databases inside of a single running instance.

For example, imagine we are a freelance developer using MongoDB to manage the data for multiple different projects: an e-commerce shop, a social media application, and a portfolio website. To compartmentalize our data, we can create a separate database for each project.

With all these databases in our MongoDB instance, how exactly would we choose and navigate around them? Fortunately, MongoDB offers us some handy commands to easily see a list of all our databases, switch databases, and confirm which database we are currently using.

First, let’s list all of our existing databases for our freelance projects. To see all of our databases, we can run the command `show dbs`. This will output a list of all the databases in our current instance and the disk space each takes up. Here is what it might look like:

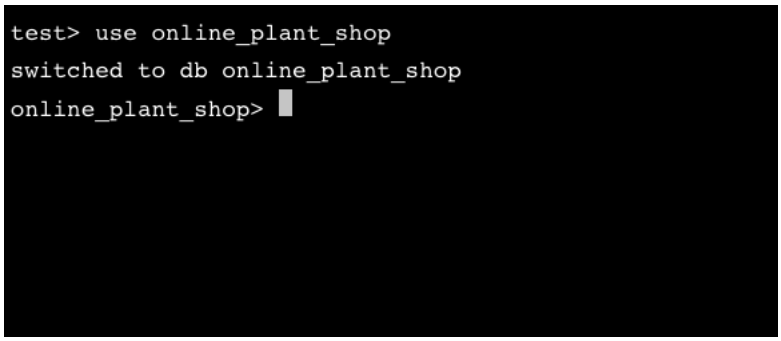
```
online_plant_shop       73.7 KiB
plant_lovers_meet       55.7 MiB
my_portfolio_site       9.57 MiB
admin                    340 KiB
local                   1.37 GiB
config                 12.00 KiB
```

Looking at the example output above, notice three unique databases: `admin`, `config`, and `local`. These databases are included by MongoDB to help configure our instance. In addition, we have our three databases for each of our freelance projects.

> Note: We won't be working with the admin, config, and local databases throughout this course, but feel free to explore them on your own!

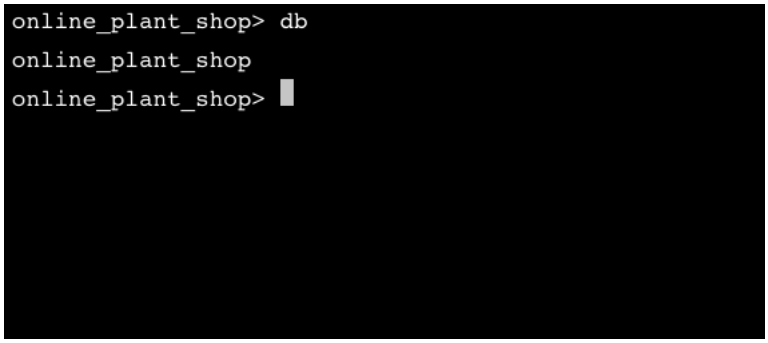
Now that we have a full list of our databases in our MongoDB instance, we will need to choose the specific one we want to work with. To navigate to a particular database, we can run the `use <db>` command. For example, if we wanted to use our e-commerce database, we'd run `use online_plant_shop`. This would place us inside our `online_plant_shop` database, where we have the option to view and manage all of its collections. It's important to note, that if the database we specify does not exist, MongoDB will create it, and place us inside of that database.

Here is what our terminal might look like:



Notice that the terminal will list the current database we are in before a `>` symbol. When we switch databases, we should see the name of the database we switched into displayed there instead. In this case, we can see the prompt changed from `test>` to `online_plant_shop>`.

If at any point we lose track of what database we are in, we can orient ourselves by running the command, `db`. This will output the name of the database we are currently using. It would look like this:



Now that we have covered the basics, let's practice navigating a MongoDB instance!

To conclude:

1. show dbs - to list out all of the current databases
2. use [database] - to use the database (ex: use restaurants)
3. db - to check if the database is the database (ex: at restaurants database uses db, it outputs restaurants, meaning it is correct)

Introduction to Querying

In the world of databases, persistence describes a database's ability to store data that is stable and enduring. There are four essential functions that a persistent database must be able to perform: *create* new data entries, and *read*, *update* and *delete* existing entries. We can summarize these four operations with the acronym CRUD.

In this lesson, we'll focus on the R in CRUD, reading data. So - how exactly do we start to read data from our MongoDB database? Well, in order to read data, we must first **query** the database. Querying is the process by which we request data from the database. The most common way to query data in MongoDB is to use the `.find()` method. Let's take a look at the syntax:

```
db.<collection>.find()
```

Notice the `.find()` method must be called on a specific collection. When we call `.find()` without arguments, it will match all of the documents in the specified collection. If our query is successful, MongoDB will return a **cursor**, an object that points to the documents matched by our query. Because our queries could potentially match large numbers of documents, MongoDB uses cursors to return our results in batches.

In other words, when we query collections using the `.find()` method, MongoDB will return up to the first set of matching documents. If we want to see the next batch of documents, we use the `it` keyword (short for iterate).

Now, let's practice using the `.find()` method!

To conclude:

1. db.<collection>.find() - to find the collection inside the database (ex: db.listingsAndReviews.find())
2. use "it" to iterate and find the next batch of collections (just type "it", will do)

Querying Collections

In the last exercise, we learned how to use MongoDB's `.find()` command to query all documents in a collection. However, what if we wanted to find a specific set of data in our collection? If we are looking for a specific document or set of documents, we can pass a query to the `.find()` method as its first argument (inside of the parenthesis `( )`). With the `query` argument, we can list selection criteria, and only return documents in the collection that match those specifications.

The query argument is formatted as a document with field-value pairs that we want to match. Have a look at the example syntax below:

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

We can have as many field-value pairs as we want in our query! To see the query in action, consider the following collection (shortened for brevity) of automobile makers in a collection named Notice the `auto_makers`:

```
{
  maker: "Honda",
  country: "Japan",
  models: [
    { name: "Accord" },
```

```
    { name: "Civic" },
    { name: "Pilot },
    …
  ]
},

{
  maker: "Toyota",
  country: "Japan",
  models: [
    { name: "4Runner" },
    { name: "Corolla" },
    { name: "Rav4" },
    …
  ]
},
{
  maker: "Ford",
  country: "USA",
  models: [
    { name: "F-150" },
    { name: "Bronco"},
    { name: "Escape"},
    …
  ]
}
```

Imagine we wanted to query this collection to find all of the vehicles that are manufactured in Now, let's practice using the `"Japan"`. We could use the `.find()` command with a query, like so:

```
db.auto_makers.find({ country: "Japan" });
```

This would output the following documents from our collection:

```
{
  maker: "Honda",
  country: "Japan",
  models: [
    { name: "Accord" },
    { name: "Civic" },
    { name: "Pilot },
    …
  ]
},
{
  maker: "Toyota",
  country: "Japan",
  models: [
    { name: "4Runner" },
    { name: "Corolla" },
```

```
    { name: "Rav4" },
    …
  ]
}
```

> Note: Query fields and their associated values are case and space sensitive. So, a query for a value "Corolla" would not be valid for a lowercase version like "corolla". This also applies if we accidentally included spaces. So, " corolla" would also not be valid if the value was "corolla".

Under the hood, `find()` is actually using an **operator** to find matches to our query. Operators are special syntax that specifies some logical action we want to perform when our method executes. In the case of the `.find()` method, it uses the implicit equality operator, `$eq`, to match documents that include the specified field and value.

If we wanted to explicitly include the equality operator in our query document, we could do so with the following field-value pair:

```
{
  <field>: { $eq: <value> }
}
```

This is the equivalent of using the format seen in the first example:

```
{
  <field>: <value>
}
```

Fortunately, MongoDB handles implicit equality for us, so we can simply use the shorthand syntax for basic queries. In the upcoming exercises, we'll learn about other operators that we can use to specify ranges and other criteria for matching documents in our queries.

Let's practice using `.find()` to do some basic querying on our `restaurants` database!

To conclude:

You can set query parameters in MongDB.

1. For example, I want to find the borough called brooklyn so I do this (db.listingsAndReviews.find({ borough: “brooklyn”})
2. If I want to set more paramaters I can do this as well (db.listingsAndReviews.find({borough: "Brooklyn", cuisine: "Caribbean"})

Querying Embedded Documents

When we are working with MongoDB databases, sometimes we'll want to draw connections between multiple documents. MongoDB lets us embed documents directly within a parent document. These nested documents are known as **sub-documents**, and help us establish relationships between our data. For example, take a look at a single record from our `auto_makers` collection:

```
{
  maker: "Honda",
  country: "Japan",
```

```
  models: [
    { name: "Accord" },
    { name: "Civic" },
    { name: "Pilot" },
    …
  ]
},
…
```

Notice how inside of this document, we have a field named `models` that nests data about a maker's specific model names. Here, we are establishing that the car maker `"Honda"` has multiple models that are associated with it. We will touch on building relationships in our database a bit later in the course, but for now, we need to know how to query them! Once again, we can use the `.find()` method to query these types of documents, by using dot notation (`.`), to access the embedded field.

Let's take a look at the syntax for querying on fields in embedded documents:

```
db.<collection>.find(
  {
    "<parent_field>.<embedded_field>": <value>
  }
)
```

Note two important parts of the syntax:

1. To query embedded documents, we must use a parent field (the name of the field wrapping the embedded document), followed by the dot (`.`) notation, and the embedded field we are looking for.
2. To query embedded documents, we must wrap the parent and embedded fields in quotation marks.

To see this in action, let's return to our previous example of the `auto_makers` collection:

```
{
  maker: "Honda",
  country: "Japan",
  models: [
    { name: "Accord" },
    { name: "Civic" },
    { name: "Pilot" },
    …
  ]
},

{
  maker: "Toyota",
  country: "Japan",
  models: [
    { name: "4Runner" },
    { name: "Corolla" },
    { name: "Rav4" },
    …
  ]
},
```

```
{
  maker: "Ford",
  country: "USA",
  models: [
    { name: "F-150" },
    { name: "Bronco"},
    { name: "Escape"},
    …
  ]
}
```

Notice, like we saw earlier, that the `model` fields contain an array of embedded documents. If we wanted to find the document with `"Pilot"` listed as a model, we would write the following command:

```
db.auto_makers.find({ "models.name" : "Pilot" })
```

This query would return the following document from our collection:

```
{
  maker: "Honda",
  country: "Japan",
  models: [
    { name: "Accord" },
    { name: "Civic" },
    { name: "Pilot" },
    …
  ]
}
```

Before moving on, let's practice querying on fields in embedded documents!

To conclude:
1. Showing database collections you can use this command (show collections .) to show all.

2. Embedded documents (has dot in between the wrapped fields), you can use commands like for example db.listingsAndReviews.find({ "address.zipcode": "11231" })

Comparison Operators: $gt and $lt

In the previous exercise, we briefly learned about MongoDB's implicit equality operator `$eq`. MongoDB provides us with many more comparison query operators that we can use to match documents based on other measures of equality. In this exercise, we'll learn how to match documents that are greater than or less than a specified value.

The greater than operator, `$gt`, is used in queries to match documents where the value for a particular field is greater than a specified value. Let's have a look at the syntax for the `$gt` operator:

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

To see the `$gt` operator in action, consider the following collection of US National Parks:

```
{
 name: "Yosemite National Park",
 state: "California",
 founded: 1890
},
{
 name: Crater Lake National Park,
 state: "Oregon",
 founded: 1902
},
{
 name: "Mesa Verde National Park",
 state: "Colorado",
 founded: 1906
},
{
 name: "Olympic National Park",
 state: "Washington",
 founded: 1909
},
…
```

To find all parks that were founded after the year 1900, we could execute the following query:

```
db.national_parks.find({ founded: { $gt: 1900 }});
```

This would return documents where To see this in action, let's return to our previous example of the `founded` is `1901` or greater:

```
{
 name: Crater Lake National Park,
 state: "Oregon",
 founded: 1902
},
{
 name: "Mesa Verde National Park",
 state: "Colorado",
 founded: 1906
},
{
 name: "Olympic National Park",
 state: "Washington",
 founded: 1909
}
```

> Note: If we wanted to include the year 1900 in our query, we could use the $gte, which will match all values that are greater than or equal to the specified value.

We can also match documents that are less than a given value, by using the less than operator, `$lt`. For example, if we reference our `national_parks` example above, we could select all parks that were founded before `1900` with the following query:

```
db.national_parks.find({ founded: { $lt: 1900 }});
```

This would return all documents where `founded` is `1899` or lower.

```
{
 name: "Yosemite National Park",
 state: "California",
 founded: 1890
}
```

> Note: Similar to the $gte operator, we can use $lte if we want to match all values that are less than or equal to the specified value.

While the examples we examined in this exercise match numerical values, it's worth noting that these comparison operators can be used with any data type (e.g., letters).

Let's practice using these operators to query our `listingsAndReviews` collection!

To conclude:

1. you can set $lt, $gt but must add { } inside to find the target value you want.

Sorting Documents

When working with a MongoDB collection, there will likely be instances when we want to sort our query results by a particular field or set of fields. Conveniently, MongoDB allows us to sort our query results before they are returned to us.

To sort our documents, we must append the The greater than operator, `.sort()` method to our query. The `.sort()` method takes one argument, a document specifying the fields we want to sort by, where their respective value is the sort order.

Take a look at the syntax for sorting a query below:

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

There are two values we can provide for the fields: `1` or `-1`. Specifying a value of `1` sorts the field in ascending order, and `-1` sorts in descending order. For datetime and string values, a value of `1` would sort the fields, and their corresponding documents, in chronological and alphabetical order, respectively, while `-1` would sort those fields in the reverse order.

Let's look at an example to see the `.sort()` method in action. Imagine we are developing an e-commerce site that sells vintage records, and our application needs to retrieve a list of inventoried records by their release year. We could run the following command to sort our records by the year they were released:

```
db.records.find().sort({ "release_year": 1 });
```

This query might return the following list of records sorted by their This would return documents where To see this in action, let's return to our previous example of the `release_year`, in ascending order.

```
{
  _id: ObjectId(...),
  artist: "The Beatles",
  album: "Abbey Road",
  release_year: 1969
},
{
  _id: ObjectId(...),
  artist: "Talking Heads",
  album: "Stop Making Sense",
  release_year: 1984
},
{
  _id: ObjectId(...),
  artist: "Prince",
  album: "Purple Rain",
  release_year: 1984
},
{
  _id: ObjectId(...),
  artist: "Tracy Chapman",
  album: "Tracy Chapman",
  release_year: 1988
}
…
```

It's important to note that when we sort on fields that have duplicate values, documents that have those values may be returned in any order. Notice in our example above, that we have two documents with the `release_year` `1984`. If we were to run this exact query multiple times, documents would get returned in numerical order by `release_year`, but the two documents that have `1984` as their `release_year` value, might be returned in a different order each time.

We can also specify additional fields to sort on to receive more consistent results. For example, we can execute the following query to sort first by We can also match documents that are less than a given value, by using the less than operator, `release_year` and then by `artist`.

```
db.records.find().sort({ "release_year": 1,  "artist": 1 });
```

This would return a list of matching documents that were sorted first by the This would return all documents where `release_year` field in ascending order. Then, within each `release_year` value, documents would be sorted by the `artist` field in ascending order. Our query result would look like this:

```
{
  _id: ObjectId(...),
  artist: "The Beatles",
  album: "Abbey Road",
  release_year: 1969
},
{
  _id: ObjectId(...),
  artist: "Prince",
  album: "Purple Rain",
  release_year: 1984
},
{
  _id: ObjectId(...),
  artist: "Talking Heads",
  album: "Stop Making Sense",
  release_year: 1984
},
{
  _id: ObjectId(...),
  artist: "Tracy Chapman",
  album: "Tracy Chapman",
  release_year: 1988
}
…
```

Notice how the two documents with the `release_year` `1984`, are now also sorted alphabetically, by the `artist` field.

To conclude:

1. Sort ascending order (Ex: db.listingsAndReviews.find({cuisine: "Spanish"}).sort({name: 1}))
2. Sort descending order (Ex: db.listingsAndReviews.find({borough: "Queens"}).sort({"address.zipcode": -1}))

Query Projections

MongoDB allows us to store some pretty large, detailed documents in our collections. When we run queries on these collections, MongoDB returns whole documents to us by default. These documents may store deeply nested arrays or other embedded documents, and because of the flexible nature of MongoDB data, each might have a unique structure. All of this complexity can make these documents a challenge to parse, especially if we're only looking to read the data of a few fields.

Fortunately, MongoDB allows us to use The greater than operator, **projections** in our queries to specify the exact fields we want to return from our matching documents. To include a projection, we can pass a second argument to the `.find()` method,

a `projection` document that specifies the fields we want to include, or exclude, in our returned documents. Fields can have a value of `1`, to include that field, or `0` to exclude it.

Let's take a closer look at the syntax for `projection` documents below:

```
db.<collection>.find(
  <query>,
  {
    <projection_field_1>: <0 or 1>,
    <projection_field_2>: <0 or 1>,
    …
  }
)
```

Consider a document from the There are two values we can provide for the fields: `listingsAndReviews` collection that we've been working with. Each document in the collection shares a similar structure to the one below:

```
{
  _id: ObjectId("5eb3d668b31de5d588f4292a"),
  address: {
    building: '2780',
    coord: [ -73.98241999999999, 40.579505 ],
    street: 'Stillwell Avenue',
    zipcode: '11224'
  },
  borough: 'Brooklyn',
  cuisine: 'American',
  grades: [
    { date: ISODate("2014-06-10T00:00:00.000Z"), grade: 'A', score: 5 },
    { date: ISODate("2013-06-05T00:00:00.000Z"), grade: 'A', score: 7 },
    {
      date: ISODate("2012-04-13T00:00:00.000Z"),
      grade: 'A',
      score: 12
    },
    {
      date: ISODate("2011-10-12T00:00:00.000Z"),
      grade: 'A',
      score: 12
    }
  ],
  name: 'Riviera Caterer',
  restaurant_id: '40356018'
}
```

You can imagine how viewing up to 20 documents just like this one, inside of a terminal, could easily get overwhelming.

If we were to query this collection and were just interested in viewing the This would return documents where To see this in action, let's return to our previous example of the `address` and `name` fields, we could run the following query that includes a projection:

```
db.listingsAndReviews.find( {}, {address: 1, name: 1} )
```

This would return the It's important to note that when we sort on fields that have duplicate values, documents that have those values may be returned in any order. Notice in our example above, that we have two documents with the `address` and `name` fields for any documents that match our query. Our output for each document would look as follows:

```
{
  _id: ObjectId("5eb3d668b31de5d588f4292a"),
  address: {
    building: '2780',
    coord: [ -73.98241999999999, 40.579505 ],
    street: 'Stillwell Avenue',
    zipcode: '11224'
  },
  name: 'Riviera Caterer'
}
```

Notice how, by default, the `_id` field is included in our projection, even if we do not specify to include it.

But what if we're not interested in seeing the This would return all documents where `_id` field? We can omit it from our results by specifying the `_id` field in our `projection` document. Instead of setting its value to `1`, we'd set it to a value of `0` to exclude it from our return documents.

```
db.listingsAndReviews.find( {}, {address: 1, name: 1, _id: 0} )
```

In some scenarios, we may need our query to return all the fields, except a select few. Rather than listing the fields we want to return, we can use a projection to define which fields we want to exclude from our matching documents by assigning the fields a value of `0`. For example, if we wanted to query our collection and see all fields *but* the `grades` field, we could run the following command:

```
db.restaurants.find( {}, {grades: 0} )
```

It is important to note that except for the `_id` field, it is not possible to combine inclusion and exclusion statements in a single projection document. For example, the following query with a projection would be invalid, and return a `MongoServerError`:

```
db.restaurants.find({}, {grades: 0, address: 1 })
```

CRUD 1 - Recap

- We can view a list of all our databases by running the `show dbs` command.
- We can navigate to a particular database, or see which database we are currently using with the `use <db>`, and `db` commands, respectively.
- We can use the `.find()` method to query a collection. Excluding a `query` argument matches all documents from the collection.

- We can match documents on particular field values by passing a `query` argument to the `.find()` method.
- When a collection's record has an embedded document, we can query the fields inside of it using dot notation (`.`) and wrapping the fields in quotation marks.
- The `$gt` and `$lt` comparison operators allow our query to match documents where the value for a particular field is greater than or less than a given value, respectively.
- We can use the `.sort()` method to sort our query results by a particular field in ascending or descending order.
- We can include a projection in our query to include or exclude certain fields from our returned documents.

Querying for An Entire Array

At this point, we should be familiar with querying in MongoDB using the `.find()` method. Let's take things a step further by learning how we can use this same method to filter documents based on array fields.

Consider a collection called `books` where each document shares a similar structure to the following:

```
{
  _id: ObjectId(...),
  title: "Alice in Wonderland",
  author: "Lewis Carroll",
  year_published: 1865,
  genres: ["childrens", "fantasy", "literary nonsense"]
}
```

Imagine we are looking for a new book to dive into – specifically, one that spans the young adult, fantasy, and adventure genres. We can query the collection for an array containing these exact values by using the `.find()` method and passing in a query argument that includes the field and array we want to match:

```
db.books.find({ genres: ["young adult", "fantasy", "adventure"] })
```

This query would return documents where the `genres` field is an array containing exactly three values, `"young adult"`, `"fantasy"`, and `"adventure"`. For example, we would get a result that might look like this:

```
{
  _id: ObjectId(...),
  title: "Harry Potter and The Sorcerer's Stone",
  author: "JK Rowling",
  year_published: 1997,
  genres: ["young adult", "fantasy", "adventure"]
},
{
  _id: ObjectId(...),
  title: "The Gilded Ones",
  author: "Namina Forna",
  year_published: 2021,
  genres: ["young adult", "fantasy", "adventure"]
}
```

```

```

Note that this query would only return documents where the array field contains precisely the values included in the query in the specified order. The following document contains the same values as mentioned in our query, but it wouldn't be matched by our search because these values are in a different order:

```
{
  _id: ObjectId(...),
  title: "Children of Blood and Bone",
  author: "Tomi Adeyemi",
  year_published: 2018,
  genres: ["fantasy", "young adult", "adventure"]
}
```

The following document would also not be matched because it contains an additional value not specified by our query:

```
{
  _id: ObjectId(...),
  title: "Eragon",
  author: "Christopher Paolini,
  year_published: 2002,
  genres: ["young adult", "fantasy", "adventure", "science fiction"]
}
```

Matching Individual Array Elements

We've just learned how to execute queries that match an entire array where values are in a specific order. This is certainly useful, but it would be incredibly limiting if this were the only way we could query arrays. Fortunately, MongoDB gives us some additional syntax that we can use with `.find()` to make our searches more flexible.

First, we'll learn how to match a single array element. Let's return to our previous example where we are searching through a collection called

Consider a collection

Consider a collection called

```
books
```

to find our next read. This was the structure we were working with:

```
{
  _id: ObjectId(...),
  title: "Alice in Wonderland",
  author: "Lewis Carroll",
  year_published: 1865,
  genres: ["childrens", "fantasy", "literary nonsense"]
}
```

Imagine we know we want to find a book that has the genre Imagine we are looking for a new book to dive into – specifically, one that spans the young adult, fantasy, and adventure genres. We can query the collection for an array containing these exact values by using the `"young adult"`, but are otherwise open to any genre. Instead of providing the entire array as a query argument, we could provide just the value we want to match, like so:

```
db.books.find({ genres: "young adult" })
```

This would return all documents that have a This query would return documents where the `genres` field that is an array that contains the value `"young adult"`, in addition to any other genres. Here is what our query might return:

```
{
  _id: ObjectId(...),
  title: "Children of Blood and Bone",
  author: "Tomi Adeyemi",
  year_published: 2018,
  genres: ["fantasy", "young adult", "adventure"]
},
{
  _id: ObjectId(...),
  title: "The Hunger Games",
  author: "Suzanne Collins",
  year_published: 2008,
  genres: ["young adult", "dystopian", "science fiction"]
},
{
  _id: ObjectId(...),
  title: "The Black Flamingo",
  author: "Dean Atta",
  year_published: 2019,
  genres: ["young adult", "coming of age", "LGBTQ"]
},
…
```

Note that every document in our result has a `genres` field that contains `"young adult"` as one of its values, in no particular order.

Matching Multiple Array Elements with $all

So far, we've learned to query an array for exact matches, or individual elements. These are two extremes: searching for a specific ordering of elements, or only matching a single element. MongoDB offers us a middle ground. We can use the `$all` operator to match documents for an array field that includes all the specified elements, without regard for the order of the elements or additional elements in the array.

For example, let's say we've finished our young adult novel and are ready to move on to something that spans the science fiction and adventure genres. We could use the Consider a collection called `$all` operator to perform this query, like so:

```
db.books.find({ genres: { $all: [ "science fiction", "adventure" ] } })
```

This query might return the following documents:

```
{
  _id: ObjectId(...),
  title: "Jurassic Park",
  author: "Michael Crichton",
  year_published: 1990,
  genres: ["science fiction", "adventure", "fantasy", "thriller"]
},
{
  _id: ObjectId(...),
  title: "A Wrinkle in Time",
  author: "Madeleine L'Engle",
  year_published: 1962,
  genres: ["young adult", "fantasy", "science fiction", "adventure"]
},
{
  _id: ObjectId(...),
  title: "Dune",
  author: "Frank Herbert",
  year_published: 1965,
  genres: ["science fiction", "fantasy", "adventure"]
},
…
```

Notice that using the This query would return documents where the `$all` operator will match documents where the given array field contains all the specified elements in *any* order, not necessarily the order provided in the query. Furthermore, our search returns documents where the `genres` array includes other elements, in addition to the ones specified in our query.

(ex: db.listingsAndReviews.find({ michelin_stars: { $all: [2018, 2019] } }))

Querying on Compound Filter Conditions

In addition to querying array fields for exact matches and individual elements, we can use comparison operators to search for documents where elements in an array field meet some condition or range of conditions.

For example, imagine we have a collection of tennis athletes, called Consider a collection called `tennis_players`, with each document having a similar structure:

```
{
  _id: ObjectId(...),
  name: "Serena Williams",
  country: "United States",
  wimbledon_singles_wins: [2002, 2003, 2009, 2010, 2012, 2015, 2016]
}
```

If we wanted to search this collection for all athletes who have been Wimbledon Singles Champions from the year 2000 onward, we could run the following query:

```
db.tennis_players.find({ wimbledon_singles_wins: { $gte: 2000 } });
```

This would return all documents where the This query would return documents where the `wimbledon_singles_wins` array has at least one element with a value of `2000` or greater. Our query result might look something like this:

```
{
  _id: ObjectId(...),
  name: "Serena Williams",
  country: "United States",
  wimbledon_singles_wins: [2002, 2003, 2009, 2010, 2012, 2015, 2016]
},
{
  _id: ObjectId(...),
  name: "Venus Williams",
  country: "United States",
  wimbledon_singles_wins: [2000, 2001, 2005, 2007, 2008]
},
{
  _id: ObjectId(...),
  name: "Roger Federer",
  country: "Switzerland",
  wimbledon_singles_wins: [2003, 2004, 2005, 2006, 2007, 2009, 2012, 2017]
},
```

We can also query based on compound conditions. Let's consider that we want to search our `tennis_players` collection to find all athletes who won a Wimbledon Singles Championship either before 1935, in the first 50 years of the championship, or after 1971, in the 50 most recent years of the tournament. We could achieve this with the following query:

```
db.tennis_players.find({ wimbledon_singles_wins: { $gt: 1971, $lt: 1935 } })
```

This might return the following set of documents:

```
{
  _id: ObjectId(...),
  name: "Suzanna Lenglen",
  country: "United States",
  wimbledon_singles_wins: [1919, 1920, 1921, 1922, 1923, 1925]
},
{
  _id: ObjectId(...),
  name: "Roger Federer",
  country: "Switzerland",
  wimbledon_singles_wins: [2003, 2004, 2005, 2006, 2007, 2009, 2012, 2017]
},
…
```

Note that this query would match documents where the array contains elements that satisfy the query conditions in some combination. One element could satisfy the greater than `1971` condition, while another could satisfy the less than `1935` condition. And if the ranges overlapped, a single element could satisfy both conditions. However, using this syntax, it is not necessary that a single element satisfies all conditions.

In the next exercise we'll learn how to filter our queries such that the matching documents have at least one array element that satisfies *all* the specified criteria.

Before moving on, let's practice querying with comparison operators on compound filter conditions!

(ex: db.listingsAndReviews.find({ michelin_stars: { $gt: 2010 } }))

(ex: db.listingsAndReviews.find({ michelin_stars: { $gt: 2015, $lt: 2010 } }))

Querying for all conditions with $elemMatch

More often than not, when we specify multiple query conditions for an array field, we'll want to match at least one array element that meets *all* the filter criteria. We can accomplish this by using another operator, `$elemMatch`.

The Consider a collection called `$elemMatch` operator is used in queries to specify multiple criteria on the elements of an array field, such that any returned documents have at least one array element that satisfies all the specified criteria.

Let's reconsider our previous example about professional tennis players to see `$elemMatch` in action. Recall that documents from this collection have the following structure:

```
{
  _id: ObjectId(...),
  name: "Serena Williams",
  country: "United States",
  wimbledon_singles_wins: [2002, 2003, 2009, 2010, 2012, 2015, 2016]
}
```

Imagine that we want to search our collection again, this time, for any athletes who have won the Wimbledon Singles Championship in the current millennium, between the years 2000 and 2019.

Our query would look something like this:

```
db.tennis_players.find({
 wimbledon_singles_wins: { $elemMatch: { $gte: 2000, $lt: 2020 } }
})
```

This would only return documents whose We can also query based on compound conditions. Let's consider that we want to search our `wimbledon_singles_wins` field is an array containing at least one element that is both greater than or equal to `2000` *and* less than `2020`. Our resulting cursor might contain the following documents:

```
{
  _id: ObjectId(...),
  name: "Pete Sampras",
  country: "United States",
```

```
  wimbledon_singles_wins: [1993, 1994, 1995, 1997, 1998, 1999, 2000]
},
{
  _id: ObjectId(...),
  name: "Serena Williams",
  country: "United States",
  wimbledon_singles_wins: [2002, 2003, 2009, 2010, 2012, 2015, 2016]
},
{
  _id: ObjectId(...),
  name: "Roger Federer",
  country: "Switzerland",
  wimbledon_singles_wins: [2003, 2004, 2005, 2006, 2007, 2009, 2012, 2017]
}
```

Note that while any matching documents *must* contain at least one value in the `wimbledon_singles_wins` array that is between the range of `2000` and `2020`, this array can also include values that fall outside that range.

(ex: db.listingsAndReviews.find({ michelin_stars: { $elemMatch: { $gte: 2005, $lte: 2010 } } }))

Querying an Array of Embedded Documents

Now that we have a grasp on the basics of querying array fields, we can tackle one more common scenario - querying embedded documents. It's common for a collection to have an array of documents rather than individual values. For instance, take our `tennis_players` collection again, but now with a slightly different structure:

```
{
  _id: ObjectId(...),
  name: "Miyu Kato",
  country: "Japan",
  wimbledon_doubles_placements:
  [{
    year: 2016,
    place: 2
  },
  {
    year: 2017,
    place: 1
  },
  {
    year: 2018,
    place: 1
  },
  {
    year: 2019,
    place: 1
  }]
}
```

In the above example, we have an array field named Let's reconsider our previous example about professional tennis players to see `wimbledon_doubles_placements` that contains documents inside of it. There are two primary ways we can query the above collection: a match on an entire embedded document or a match based on a single field.

First, let's see how we can do an exact match on the *entire* embedded document. For example, if we wanted to query our `tennis_players` collection for players who placed 2nd in 2019:

```
db.tennis_players.find( { "wimbledon_doubles_placements": { year: 2019, place: 2 } } )
```

In the above query, the field order must be exactly the order we are looking for, with the exact field values. This query would match the below document because the order and values are exactly the same as the one in the query:

```
{
  _id: ObjectId(...),
  name: "Gabriela Dabrowski",
  country: "Canada",
  wimbledon_doubles_placements:
  [{
    year: 2019,
    place: 2
  }]
},
{
  _id: ObjectId(...),
  name: "Yifan Xu",
  country: “China”,
  wimbledon_doubles_placements:
  [{
    year: 2019,
    place: 2
  }]
}…
```

However, a query like this:

```
db.tennis_players.find( { "wimbledon_doubles_placements": {place: 2,  year: 2019 } } )
```

Would not return any results since there would be no documents with that specific ordering.

We can also query based on a single field. For example, if we just wanted to query for any Wimbledon doubles winners in the year 2016, we can do the following:

```
db.tennis_players.find( { "wimbledon_doubles_placements.year": 2016 } )
```

Notice that the syntax is exactly the same as when we were querying for non-array fields. The embedded document field and parent document field must be wrapped in quotation marks (single or double) and use the dot (`.`) notation. This query would return results like the following:

```
{
  _id: ObjectId(...),
  name: "Tímea Babos",
  country: "Hungary",
  wimbledon_doubles_placements:
  [{
    year: 2015,
    place: 4
   },
   {
    year: 2016,
    place: 2
  }]
{
  _id: ObjectId(...),
  name: "Yaroslava Shvedova",
  country: "Kazakhstan",
  wimbledon_doubles_placements:
  [{
    year: 2010,
    place: 1
   },
   {
    year: 2016,
    place: 2
  }]
}…
```

Here, our query result has all the documents that have an embedded document with a `year` field with the value of `2016`.

It's important to note we can even combine these queries with query operators and even do multiple query conditions using `$elemMatch`. For more examples, take a look at the official MongoDB documentation on querying embedded documents in arrays.

(ex: db.listingsAndReviews.find({ grades: {date: ISODate("2014-07-11T00:00:00.000Z"), grade: 'A', score: 8} }))

(ex: db.listingsAndReviews.find({ "grades.grade": "C" }))

Recap:

- We can query documents for exact array matches by using the `.find()` method and passing in a query document containing the field name, and its array as the value.
- We can match a single array element by using the `.find()` method and passing in a query document containing the field name, and the element we want to match as its value.
- To match multiple elements in an array, we can apply the `$all` operator to the `.find()` method.
- The `$all` operator will match any document where the given array field contains all the specified values, in any order and regardless of other elements in the array.

- We can use the `.find()` method with comparison operators to match documents where the array contains one or more elements that satisfy the query conditions in some combination.
- To match documents where the array contains one or more elements that satisfy *all* the query conditions, we can apply the `$elemMatch` operator to the `.find()` method.
- We can query embedded documents in an array field by querying for either an exact match (with the exact order) or by querying for a single field.

CRUD 2: INSERTING AND UPDATING

The _id Field

By this point, we've learned the fundamentals of querying data in MongoDB. In this lesson, we'll learn how to insert new documents, and update existing ones.

As you continue to work with documents in MongoDB, you may notice one field that exists across every document: the `_id` field. It might look similar to this:

```
_id: ObjectId("5eb3d668b31de5d588f4305b")
```

The `_id` field plays a vital role in every document inside of a MongoDB collection, and it has a few distinct characteristics:

- The `_id` field is required for every document in a collection and must be unique.
- MongoDB automatically generates an `ObjectId` for the `_id` field if no value is provided.
- Developers can specify the `_id` with something other than an `ObjectId` such as a number or random string, if desired.
- The `_id` field is immutable, and once a document has an assigned `_id`, it cannot be updated or changed.

The `ObjectId` is a 12-byte data type that is commonly used for the `_id` field. When generated automatically, each `ObjectId` contains an embedded timestamp which is generally unique. This allows documents to be inserted in order of creation time (or very close to it) and for users to roughly sort their results by creation time if necessary. While we won't explicitly need the _id field to update or create new documents, it's important to note that this is how MongoDB identifies each unique document that is inserted or updated in a collection.

Let's return to our `restaurants` collection to look at the `_id` fields that currently exist in our documents.

Inserting a Single Document

Now that we know how MongoDB identifies each individual document, let's focus on the *Create* aspect of *CRUD* operations. Specifically, how do we start adding new documents to our collections? In MongoDB, we can use the `.insertOne()` method to insert a single new document!

The syntax for the method looks as follows:

```
db.<collection>.insertOne(
  <new_document>,
  {
    writeConcern: <document>,
  }
);
```

The `.insertOne()` method has a single required parameter, the document to be inserted, and a second optional parameter named `writeConcern`. We won't be working with the `writeConcern` parameter in this course, but more details about the parameter can be found in the official MongoDB documentation. For now, know that it's an optional parameter that allows us to specify how we want our write requests to be acknowledged by MongoDB.

Let's take a look at an example where we insert a single document into a `videogames` collection:

```
db.videogames.insertOne({
  title: "Elden Ring",
  year: 2022,
  company: "Fromsoftware"
});
```

When a document is successfully inserted with `.insertOne()`, the output is an object with a field named `acknowledged` (related to the `writeConcern` parameter we mentioned earlier) with the value `true` and a field named `insertedId` where the value is the `_id` field for the newly inserted document. Here is what it looks like:

```
{
  "acknowledged": true,
  "insertedId": ObjectId("5fd989674e6b9ceb8665c63d")
}
```

Note that if we try to insert into a specified collection that does not exist, MongoDB will create one and insert the document into the newly created collection.

Now, let's practice using the `.insertOne()` method by returning to our `listingsAndReviews` collection inside the `restaurants` database.

Inserting Multiple Documents

We have just used `.insertOne()` to insert a single document into a collection, but what if we want to insert multiple documents into a collection? Here's where the MongoDB `.insertMany()` method comes in.

As its name suggests, `.insertMany()` will insert multiple documents into a collection. Much like `.insertOne()`, if the collection we've specified does not exist, one will be created.

The syntax for the method is as follows:

```
db.<collection>.insertMany(
  [<document1>, <document2>, ...],
  {
    writeConcern: <document>,
    ordered: <boolean>
  }
);
```

This method has three parameters:

1. An array of documents; the documents we want to add to the collection.
2. A parameter named `writeConcern`.
3. A parameter named `ordered`.

The When a document is successfully inserted with `ordered` parameter can be handy since it allows us to specify if MongoDB should perform an ordered or unordered insert. If set to `false`, documents are inserted in an unordered format. If set to `true`, the default behavior, MongoDB will insert the documents in the order given in the array.

It's worth noting that with ordered inserts, if a single document fails to be inserted, the entire insert operation will cease, and any remaining documents will not be inserted. On the other hand, unordered inserts will continue in the case of an insert failure and attempt to insert any remaining documents.

Let's look at an example of `.insertMany()` on a collection named `students`:

```
db.students.insertMany([
  {
    name: "Mia Ramirez",
    age: 15
  },
  {
    name: "Kiv Huang",
    age: 17
  },
  {
    name: "Sam Soto",
    age: 16
  }
], { ordered: true })
```

The command above will insert the documents in the order specified. Below you will find the output of the command with an additional note that indicates which `_id` represents which document.

```
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId("63054a5376742c0e5a0cfafb"), // Mia
    '1': ObjectId("63054a5376742c0e5a0cfafc"), // Kiv
    '2': ObjectId("63054a5376742c0e5a0cfafd") // Sam
  }
}
```

Let's practice adding multiple new restaurants to our `listingsAndReviews` collection!

Updating a Single Document

Now that we have explored a few *Create* operations, let's look at operations that *Update* data.

In MongoDB, we can use the As its name suggests, `.updateOne()` method to update a single document. The method finds the first document that matches specific filter criteria and applies specified update modifications. Note that it updates the *first* matching document, even if multiple documents match the criteria.

Let's take a look at the syntax for the `.updateOne()` method:

```
db.<collection>.updateOne(<filter>, <update>, <options>)
```

The method has three parameters:

- `filter`: A document that provides selection criteria for the document to update.
- `update`: A document that specifies any modifications to be applied. This parameter gives us quite a bit of flexibility, allowing us to modify existing fields, insert new ones, or even replace an entire document.
- `options`: A document that includes any additional specifications for our update operation such as `upsert` and `writeConcern`.

To explore the importance of each of these parameters and how the The When a document is successfully inserted with `updateOne()` method works, consider a third-party retail store for used smartphones. The store keeps all their information in a collection called `products`, where each document holds information regarding a specific type of smartphone:

```
{
  _id: ObjectId("507f1fg7bcf865d799439h11"),
  title: "iPhoneX",
  price: 1000,
  stock: 25
},
{
  _id: ObjectId("507f1fg7bcf865d799439h12"),
  title: "iPhone7",
  price: 600,
  stock: 25
},
{
  _id: ObjectId("507f1fg7bcf865d799439h13"),
  title: "iPhone6",
  price: 500,
  stock: 25
}
```

To start an update operation, we must first choose our Let's look at an example of `filter`. This is similar to when we used `find()` to retrieve a document based on specific criteria. So, for example, if we wanted to update only the document with the title `"IPhoneX"`, we could specify the title as the filter:

```
db.products.updateOne({ title: "iPhoneX" }, <update>, <options> });
```

Now that we have a document we can target for the update, we can move onto the The command above will insert the documents in the order specified. Below you will find the output of the command with an additional note that indicates which `update` parameter.

To update a document in MongoDB, we have to specify what fields we want to update and *how* we want to update them. This is where the `update` parameter comes into play. To specify how we want to update a document, we can use MongoDB update

operators. MongoDB offers us several update operators that can perform a variety of modifications to document fields. In this exercise, we'll focus on the `$set` update operator. This operator allows us to replace a field's value with one that we provide.

To see this in action, imagine a new phone model is launching soon, and the price of the Let's practice adding multiple new restaurants to our `"iPhoneX"` will need to be decreased in order to remain competitive. We want to update the price from `1000` to `679`. We can accomplish this by running the following command:

```
db.products.updateOne({ title: "iPhoneX" }, { $set: { price: 679 } });
```

If successful, the operation should return:

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

After running the previous command, we could query for the item to confirm the update was successful:

```
db.products.findOne({ title: "iPhoneX" })
```

And find our document was updated:

```
{
  _id: ObjectId("507f1fg7bcf865d799439h11"),
  title: "iPhoneX",
  price: 679
}
```

In this case, querying on the `title` field works fine, assuming the value is unique for every document. Usually, we want to be as specific as possible with our filtering criteria, so we can include multiple fields to add more specificity to our search. Remember that even if multiple documents match the filter criteria, only a single one (the first match) will be updated.

> Note: While exploring the updateOne() command, we didn't cover the use of the <options> parameter. This is because these fields are optional and aren't required to perform the base action of updating a record. To explore the <options> parameter further, visit the MongoDB documentation for the updateOne() method.

Let's practice updating using the `updateOne()` method by returning to our `restaurants` database.

Updates on Embedded Documents and Arrays

While updating a single document using a single field is helpful, MongoDB also stores data inside of embedded documents. So, what if we want to update a specific field in an embedded document? Consider the following document within a collection named `furniture`:

```
{
  _id: ObjectId("3ldh1fg733kf65d7994393ld"),
  name: "bedframe",
```

```
  dimensions: {
    length: 75,
    width: 38
  }
}
```

Let's say we wanted to update the Let's take a look at the syntax for the `width` field inside the `dimensions` embedded document. We could run the following command:

```
db.furniture.updateOne(
  { name: "bedframe" },
  { $set: { "dimensions.width": 54 }}
);
```

We can successfully target the `width` field inside the `dimensions` embedded document using dot notation.

MongoDB also stores data inside of arrays! If we instead want to update a value within an array, we can use dot notation to access the index of the element we want to update. Let's look at the following example document for a collection named `nbateams`:

```
{
  _id: ObjectId("402h1fg73cf865d799439k42"),
  team: "Chicago Bulls",
  championships: [1991, 1902, 1993, 1996, 1997, 1998]
}
```

If we want to update the 2nd element (1902) of the `championships` array to the correct year, `1992`, we could run the following command:

```
db.nbateams.updateOne(
  { team: "Chicago Bulls" },
  { $set: {"championships.1": 1992 }}
)
```

Once again, the embedded document's name and the array index must be wrapped in quotations for the command to be successful. Note that we're using the index of `1` since the year `1902` is the second element of the array, and arrays start at index `0`.

Updating an Array with New Elements

MongoDB provides different array update operators that we can use with the `.updateOne()` method. In the earlier exercises, we learned to use the `$set` operator to replace a value. In this exercise, we'll look at the `$push` operator.

The `$push` operator adds (or "pushes") new elements to the end of an array. It can be used with the `.updateOne()` method with the following syntax:

```
db.<collection>.updateOne(
  <filter>,
```

```
  { $push: { <field1>: <value1>, ... } }
);
```

Consider a collection, `automobiles`, holding a document with data regarding specific car models:

```
{
  _id: ObjectId("627934bbfd6a8619040cc287"),
  make: "Audi",
  model: "A1",
  year: [2017, 2019]
}
```

If we wanted to add a new year into the array, we could use the MongoDB also stores data inside of arrays! If we instead want to update a value within an `$push` operator to accomplish this:

```
db.vehicles.updateOne(
  { make: "Audi" },
  { $push: { year: 2020 }}
);
```

The updated document would look as follows:

```
{
  _id: ObjectId("627934bbfd6a8619040cc287"),
  make: "Audi",
  model: "A1",
  year: [2017, 2019, 2020]
}
```

It’s important to note that if the mentioned field is absent in the document to update, the Once again, the embedded document’s name and the array index must be wrapped in quotations for the command to be successful. Note that we’re using the index of `$push` operator adds this field to the document as an array and includes the given value as its element.

For example, consider our document from the Let’s practice updating embedded fields and arrays in our The command above will insert the documents in the order specified. Below you will find the output of the command with an additional note that indicates which `automobiles` collection again. Let’s say we wanted to update the country that manufactures this make of vehicle using the following command:

```
db.vehicles.updateOne(
  { make: "Audi" },
  { $push: { country: "Germany" }}
);
```

Because our document did not previously have a field called `country`, running this command would add the new field to the document as an array and insert one element, the string `"Germany"`. Our output would look as follows:

```
{
  _id: ObjectId("627934bbfd6a8619040cc287"),
  make: "Audi",
  model: "A1",
  year: [2017, 2019, 2020],
  country: [ "Germany" ]
}
```

```
{
  _id: ObjectId("627934bbfd6a8619040cc287"),
  make: "Audi",
  model: "A1",
  year: [2017, 2019, 2020],
  country: [ "Germany" ]
}
```

Upserting a Document

In the previous exercises, we looked at different ways to update and insert data into a collection. Now, we'll learn about combining both operations using `upsert`.

The `upsert` option is an optional parameter we can use with update methods such as `.updateOne()`. It accepts a boolean value, and if assigned to `true`, `upsert` will give our `.updateOne()` method the following behavior:

1. Update data if there is a matching document.
2. Insert a new document if there's no match based on the query criteria.

Let's take a look at its syntax below:

```
db.<collection>.updateOne(
  <filter>,
  <update>,
  { upsert: <boolean> }
);
```

The If we wanted to add a new year into the array, we could use the MongoDB also stores data inside of arrays! If we instead want to update a value within an `upsert` parameter is false by default. If the property is omitted, the method will only update the documents that match the query. If no existing documents match the query, the operation will complete without making any changes to the data.

To see the `upsert` option in action, consider a collection named `pets`, holding a large number of documents with the following structure:

```
{
  _id: ObjectId(...),
  name: "Luna",
  type: "Cat",
  age: 2
}
```

Imagine it's Luna's birthday, and we want to be sure that we capture her current age, but we aren't sure whether or not we have an existing document for her. This would be an excellent opportunity to use `upsert` since one of two things will happen:

- If Luna does not exist in the database, our command will create the document
- If Luna does exist, the document will be updated

To Let's practice updating embedded fields and arrays in our The command above will insert the documents in the order specified. Below you will find the output of the command with an additional note that indicates which `upsert` our document for Luna, we can call the `.updateOne()` command as follows:

```
db.pets.updateOne(
  { name: "Luna", type: "Cat"},
  { $set: { age: 3 }},
  { upsert: true }
)
```

As noted, this command would search our `pets` collection for a document where the `name` is `"Luna"` and the `type` is `"Cat"`. If such a document exists, its `age` field would be updated. Otherwise, the following document would be inserted into our collection:

```
{
  _id: ObjectId(...),
  name: "Luna",
  type: "Cat",
  age: 3
}
```

Updating Multiple Documents

So far, we have learned how to insert and update individual documents in a collection, but what if we want to update multiple documents simultaneously? This is where the MongoDB `.updateMany()` method comes in handy.

The `.updateMany()` method allows us to update all documents that satisfy a condition. The `.updateMany()` method looks and behaves similarly to `.updateOne()`, but instead of updating the first matching document, it updates all matching documents:

Let's examine its syntax:

```
db.<collection>.updateMany(
  <filter>,
  <update>,
  <options>
);
```

Like before, we have three main parameters:

1. `filter`: The selection criteria for the update.

2. `update`: The modifications to apply.
3. `options`: Other options that could be applied, such as `upsert`.

Let's see how we can apply the method to an example. Suppose a company sets a minimum salary for all employees. We want to update all employees with a salary of $75,000 to the new minimum of $80,000. Here is what our collection of To see the `employees` might look like:

```
{
  _id: ObjectId(...),
  name: "Rose Nyland",
  department: "Information Technology",
  salary: 75000
},
{
  _id: ObjectId(...),
  name: "Dorothy Zbornak",
  department: "Human Resources",
  salary: 75000
},
{
  _id: ObjectId(...),
  name: "Sophia Petrillo",
  department: "Human Resources",
  salary: 75000
},
{
  _id: ObjectId(...),
  name: "Blanche Devereaux",
  department: "Sales",
  salary: 80000
}
```

We can use Imagine it's Luna's birthday, and we want to be sure that we capture her current age, but we aren't sure whether or not we have an existing document for her. This would be an excellent opportunity to use `.updateMany()` to target all employees with the same salary in order to increase it:

```
db.employees.updateMany(
  { salary: 75000 },
  { $set: { salary: 80000 }}
)
```

In the above example, we're using the salary as the filter criteria: Let's practice updating embedded fields and arrays in our The command above will insert the documents in the order specified. Below you will find the output of the command with an additional note that indicates which `{ salary: 75000}`. This targets any document with the salary set to `75000`. We can then use the second parameter (via the `$set` operator) to update the specified fields in those documents. The collection would now look like this:

```
{
  _id: ObjectId(...),
  name: "Rose Nyland",
  department: "Information Technology",
```

```
  salary: 80000
},
{
  _id: ObjectId(...),
  name: "Dorothy Zbornak",
  department: "Human Resources",
  salary: 80000
},
{
  _id: ObjectId(...),
  name: "Sophia Petrillo",
  department: "Human Resources",
  salary: 80000
},
{
  _id: ObjectId(...),
  name: "Blanche Devereaux",
  department: "Sales",
  salary: 80000
}
```

Notice how all employees with the `salary` of `75000` had their salary increased to `80000` while the employee whose `salary` was already `80000` stayed the same.

Modifying Documents

In MongoDB, sometimes we may want to update a document but also return the document we modified. The So far, we have learned how to insert and update individual documents in a collection, but what if we want to update multiple documents simultaneously? This is where the MongoDB `.findAndModify()` method modifies and *returns* a single document. By default, the document it returns does not include the modifications made on the update. This method can be particularly useful if we want to see (or use) the state of an updated document after we perform an update operation. This method also has a lot of flexible optional parameters that aren't available in other methods.

Let's take a look at the syntax of the The `.findAndModify()` method below:

```
db.<collection>.findAndModify({
  query: <document>,
  update: <document>,
  new: <boolean>,
  upsert: <boolean>,
  ...
});
```

Note that there are four commonly used fields:

1. `query`: Defines the selection criteria for which record needs modification.
2. `update`: A document that specifies the fields we want to update and the changes we want to make to them.
3. `new`: When `true`, this field returns the modified document rather than the original.

4. `upsert`: Creates a new document if the selection criteria fails to match a document.

> Note: In addition to these fields, the .findAndModify() method accepts many other options. We will not be covering them here, but more details can be found in the documentation for the .findAndModify() method.

With this method, there are a lot of scenarios that can occur. First, let's consider a collection called Let's see how we can apply the method to an example. Suppose a company sets a minimum salary for all employees. We want to update all employees with a salary of $75,000 to the new minimum of $80,000. Here is what our collection of `foodTrucks` containing the following document:

```
{
  _id: ObjectId(...),
  name: "Criff Dogs",
  address: "15 Bedford Ave",
  shutdown: false
},
{
  _id: ObjectId(...),
  name: "Sals Pizza",
  address: "249 Otter Place",
  shutdown: false
}
```

The first scenario we can observe is if we wanted to update a document and see the updated document state pre-modification (before it was changed). This is the default behavior of the method. So, if we were to change the We can use `shutdown` property of the document above with the name `"Criff Dogs"`, we can run the following command:

```
db.foodTrucks.findAndModify({
  query:  { name: "Criff Dogs" },
  update: { shutdown: true }
});
```

The output of this method would be the document In the above example, we're using the salary as the filter criteria: *before* it was modified. Notice the `shutdown` field is still false, even though we changed it to `true`:

```
//Output
{
  _id: ObjectId(...),
  name: "Criff Dogs",
  address: "15 Bedford Ave",
  shutdown: false
}
```

The next scenario is similar but would use the Notice how all employees with the `new` field to return a different output. If we ran the following query:

```
db.foodTrucks.findAndModify({
  query:  { name: "Criff Dogs" },
  update: { shutdown: true },
  new: true
});
```

We would be able to see the *new modified* document as the output. Notice the value in the output for the `shutdown` field is `true`:

```
//Output
{
  _id: ObjectId(...),
  name: "Sals Pizza",
  address: "249 Otter Place",
  shutdown: true
}
```

Lastly, we can use the `upsert` field to be able to add documents if they do not currently exist in the database. So if we ran the following command:

```
db.foodTrucks.findAndModify({
  query:  { name: "Ben and Jerry", address: "17 Cliff Pl" },
  update: { shutdown: false },
  new: true,
  upsert: true
});
```

MongoDB would then search the collection `foodTrucks` based on the query argument, and if it didn't find a match, it would create the document. So our new food truck would be added to the collection, and our return output would be:

```
//Output
{
  _id: ObjectId(...),
  name: "Ben and Jerry",
  address: "17 Cliff Pl",
  shutdown: false
}
```

We might notice that `.updateOne()` and `.findAndModify()` behave quite similarly. Both will update a document in our database or create one if it doesn't exist. So what are the main differences? Well, `.findAndModify()` returns the document that you modify, whereas `.updateOne()` does not. Moreover, `.findAndModify()` allows us to specify whether we want to return the old or new (modified version) of the updated document with the use of the `new` parameter.

Key Takeaway

- The `_id` field is a unique identifier for documents in a collection. By default, MongoDB assigns an `ObjectId` value to the `_id` field for each document.
- Individual documents can be added to a collection with `.insertOne()`, and the document to be inserted is provided as an argument.
- Multiple documents can be inserted into a collection with the `.insertMany()` method. An array containing all the documents to insert is passed in an argument.
- The `.updateOne()` method is used to update the first document within the collection that matches a given query.
- We can use `.updateMany()` to update multiple matching documents simultaneously.
- The `$push` operator appends a specified value to an array.
- The `.findAndModify()` method modifies and returns a single document in a collection. By default, it returns the original document, and if no matching document is found, a new one can be inserted by adding the upsert option.

In addition to the methods we've learned throughout this lesson, MongoDB offers us other syntax and commands that can be useful when inserting, updating, or replacing documents:

- The `ordered` parameter can be provided to the `.insertMany()` method. It accepts a boolean value, and, if set to false, will insert the documents in an unordered format to increase performance.
- The `$unset` operator can be provided to the `.updateOne()` or `.updateMany()` method. It removes a particular field from a document.
- The `.findOneAndUpdate()` method is similar to `.updateOne()`, but instead of returning a document acknowledging the success or failure of our operation, it returns either the original or updated document.
- The `.renameCollection()` method allows us to update the name of our collection without modifying any of its documents.
- The `.bulkWrite()` method allows us to perform multiple write operations (updating or inserting) with controls for order of execution.

There are many different implementations of inserting or modifying data in MongoDB, so make sure to explore the docs for more examples!

Deleting a Document

So far we have learned how to **C**reate, **R**ead, and **U**pdate data using different methods provided by MongoDB. Let's explore the last CRUD operation, **D**elete.

There are certain situations when data is no longer necessary or becomes obsolete. MongoDB provides us a couple options to permanently remove unwanted documents from a collection. In this exercise, we'll focus on learning how to use the `.deleteOne()` method to remove a single document from a collection.

In order to use `.deleteOne()`, we must provide specific filtering criteria to find the document we want to delete. MongoDB will look for the first document in the collection that matches the criteria and delete it. Let's take a closer look at the syntax:

```
db.<collection>.deleteOne(<filter>, <options>);
```

Note in the syntax above, the `.deleteOne()` method takes two arguments:

- `filter`: A document that provides selection criteria for the document to delete.
- `options`: A document where we can include optional fields to provide more specifications to our operation, such as a `writeConcern`.

To see `.deleteOne()` in action, consider a collection, `monsters`, with the following documents:

```
{
  _id: ObjectId(...),
```

```
  name: "Luca",
  age: 100,
  type: "Hydra"
},
{
  _id: ObjectId(...),
  name: "Lola",
  age: 95,
  type: "Hydra"
},
{
  _id: ObjectId(...),
  name: "Igor",
  age: 95,
  type: "Chimera"
},
```

If we want to delete a single monster with an age of `95`, we can run the following command:

```
db.monsters.deleteOne({ age: 95 });
```

If the command is successful, MongoDB will confirm the document was deleted with the following output:

```
{ acknowledged: true, deletedCount: 1 }
```

The collection would then be left with these remaining documents:

```
{
  _id: ObjectId(...),
  Name: "Luca",
  age: 100,
  type: "Hydra"
},
{
  _id: ObjectId(...),
  name: "Igor",
  age: 95,
  type: "Chimera"
},
```

Notice that only one of the documents in the collection with the `age` of `95` was deleted. When the filter criteria is non-unique, the document that gets deleted is the first one that MongoDB identifies when performing the operation. Which document is found first depends on several factors which can include insertion order and the presence of indexes relevant to the filter.

Deleting Multiple Documents

We now know how to delete a single document from a collection, but what if we want to delete multiple documents that match certain criteria? We can accomplish this with the `.deleteMany()` method.

The `.deleteMany()` method removes all documents from a collection that match a given filter. This method uses the following syntax:

```
db.<collection>.deleteMany(<filter>, <options>);
```

Note in the syntax above that the `.deleteMany()` method takes two arguments:

- `filter`: A document that provides selection criteria for the documents to delete.
- `options`: A document where we can include optional fields to provide more specifications to our operation, such as a `writeConcern`.

> Warning: If no filter is provided to the .deleteMany() method, all documents from the collection will be deleted.

Let's revisit the original `monsters` collection from the previous exercise. Consider a new monster with the name of `"Pat"` was recently added:

```
{
  _id: ObjectId("629a1e8c2bf029cc101c92d4"),
  name: "Luca",
  age: 100,
  type: "Hydra"
},
{
  _id: ObjectId("629a2245b8bd9cad32a210fa"),
  name: "Lola",
  age: 95,
  type: "Hydra"
},
{
  _id: ObjectId("629a225119915a53df5b428c"),
  name: "Igor",
  age: 85,
  type: "Hydra"
},
{
  _id: ObjectId("629a226c8982a4dd04e093ff"),
  name: "Pat",
  age: 85,
  type: "Dragon"
}
```

We now want to get rid of all the monsters with a `type` field with the value of `"Hydra"`. We could run the `.deleteOne()` method and pass in the filter `{type: "Hydra"}`, but we would need to execute the method multiple times. This could quickly get very tedious. Instead, let's use `.deleteMany()`:

```
db.monsters.deleteMany({ type: "Hydra" });
```

Once executed, the

Once executed, the operation will successfully delete all documents where the

```
type
```

field has the value of

```
"Hydra"
```

. MongoDB will confirm if the operation was successful and let us know how many documents were deleted with the following output:

```
{ acknowledged: true, deletedCount: 3 }
```

This would leave us with a single remaining document:

```
{
  _id: ObjectId("629a226c8982a4dd04e093ff"),
  name: "Pat",
  age: 85,
  type: "Dragon"
}
```

Replacing a Document

We might encounter situations where we need to delete a document and immediately replace it with another one. We could achieve this by running two separate methods, `.deleteOne()`, then `.insertOne()`. Fortunately, MongoDB provides us with a single method, `.replaceOne()`, that can both delete and insert all at once!

The `.replaceOne()` method replaces the first document in a collection that matches the given filter. The syntax for this method looks as follows:

```
db.<collection>.replaceOne(
  <filter>,
  <replacement>,
  <options>
);
```

Note in the syntax above that the `.replaceOne()` method takes three arguments:

- `filter`: A document that provides selection criteria for the document to replace.
- `replacement`: The replacement document.

- `options`: A document where we can include optional fields to provide more specifications to our operation, such as `upsert`.

The replacement document can contain a subset of fields of the original document or entirely unique fields. To see it in action, consider a collection named `employees` with the following documents:

```
{
  _id: ObjectId(...),
  name: "Rohit Kohli",
  department: "Customer Analytics"
  position: "Senior Software Engineer"
},
{
  _id: ObjectId(...),
  name: "Rin Paterson",
  department: "People Operations",
  position: "Head of People Operations"
}
```

Imagine `"Rohit Kohli"` leaves the company. We still want to store their name, and update their position to `"N/A"` but no longer need a `department` field and value. We'd then need to replace Rohit's current document with a new document that only contains two fields: `name`, and `position`. We can accomplish this with `.replaceOne()`:

```
db.employees.replaceOne(
  { name: "Rhoit Kohli" },
  { name: "Rohit Kohli", position: "N/A" }
);
```

If successful, this would produce the following output:

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

After running this command, we'd be left with the following collection:

```
{
  _id: ObjectId(...),
  name: "Rohit Kohli",
  position: "N/A"
},
{
  _id: ObjectId(...),
  name: "Gary Paterson",
  department: "HR",
```

```
  position: "Head of Human Resources"
}
```

Notice how the other fields were completely removed. This is the key difference between `.replaceOne()` and `.updateOne()`. Whereas `.updateOne()` updates specific fields based on the update modifiers provided, `.replaceOne()` replaces the entire document and will only include fields specified in the `<replacement>` parameter.

Key Takeaway

- The `.deleteOne()` method deletes a single document from a collection. It accepts a filter document specifying which document to delete as the first parameter.
- The `.deleteOne()` method will only delete the *first* matching document in the collection.
- The `.deleteMany()` method deletes all matching documents from a collection. It accepts a filter document specifying which document to delete as the first parameter.
- The `.replaceOne()` method replaces an entire document from a collection. It takes in filtering criteria specifying the document to replace as the first parameter and a replacement document as the second one.
- The `.replaceOne()` method will only replace the *first* matching document in the collection.
- Since `.replaceOne()` replaces an entire document, only fields included in the second parameter will be present in the document after the operation executes.

In addition to the syntax we've learned throughout this lesson, MongoDB offers us other syntax and commands that can be useful when deleting or replacing documents:

- The `.findOneAndReplace()` method is very similar to `.replaceOne()`. It replaces a document in a collection based on filter criteria, but instead of returning a document that acknowledges the operation, it returns either the original document or the replacement document.
- The `.findOneAndDelete()` method deletes a document, and returns the deleted document.

INTRODUCTION TO INDEXING

In MongoDB, indexes play an important role in making sure our database performs optimally. Recall, an index is a special data structure that stores a small portion of the collection's data in an easy-to-traverse form. We have already used indexes when we queried by the `_id` field since MongoDB creates a default index on the `_id` field for all our documents.

However, we can also create our own custom index by using the `.createIndex()` method. The syntax looks like this:

```
db.<collection>.createIndex({ <keys>, <options>, <commitQuorum>)}
```

We have three main parameters:

- `keys`: A document that contains the field and value pairs where the field is the index key and the value describes the type of index for that field.
- `options`: A document of various optional options that control index creation.

- `commitQuoroum`: A more advanced parameter that gives control over replica sets. We won't be working with this parameter in this lesson.

In this lesson, we will mostly work with the `keys` parameter. To learn more about the various other parameters, visit the official documentation. That said, our syntax will look closer to this:

```
db.<collection>.createIndex({ <field>: <type> });
```

For the `keys` parameters, we must pass a document with field-type pairs. Fields can be assigned a value of `1` or `-1`. A value of `1` will sort the index in ascending order, while a value of `-1` would sort the index in descending order. If the field contains a string value, `1` will sort the documents in alphabetical order (A-Z), and `-1` will sort the documents in reverse order (Z-A).

To see `.createIndex()` in action, imagine as a university president, we have a collection of every student within your database, called `students`. A sample document from the `students` collection might look like this:

```
{
  _id: ObjectId(...),
  last_name: "Tapia",
  first_name: "Joseph",
  major: "architecture",
  birth_year: 1988,
  graduation_year: 2012,
  year_abroad: 2011
}
```

Perhaps we are organizing reunions for students who studied abroad and found ourselves frequently searching the database for students who studied internationally during a particular year. Rather than repeatedly querying the entire `students` collection by the `year_abroad` field, it would be beneficial to create an index based on this particular field, also known as a single field index. We could run the following command:

```
db.students.createIndex({ year_abroad: 1 });
```

The above command would create an index on all the documents in the student's collection based on the `year_abroad` field, sorted in ascending order.

We can run a query on the indexed field to utilize the indexed `year_abroad` field. Here's an example query that uses this new index to search for students who have studied abroad from 2020 onward:

```
db.students.find({ year_abroad: { $gt: 2019 }});
```

Creating a single field index can save us significant time in our query since we'd only be scanning the index for ordered values of the `year_abroad` field rather than browsing the entire collection and examining every document.

Performance Insights with .explain()

Since indexing in MongoDB is tied closely to In MongoDB, indexes play an important role in making sure our database performance, it would be ideal to have a way to see how our indexes impact our queries. The `.explain()` method can offer us insight into the performance implications of our indexes. The method has the following syntax:

```
db.<collection>.find(...).explain(<verbose>)
```

Note that the method is appended to the `.find()` method. It also takes one string parameter named `verbose` that specifies what the method should explain. The possible values are: `"queryPlanner"`, `"executionStats"`, and `"allPlansExecution"`. Each value offers meaningful insights on a query. To gain insights regarding the execution of the winning query plan for a query, we can use the `"executionStats"` option.

To see `.explain()` in action, let's refer back to our study abroad example from the previous exercise. Let's examine how to use this method by appending the `.explain()` method to our query from the previous exercise:

```
db.students.find({ year_abroad: { $gt: 2019 }}).explain('executionStats');
```

Running our query with In this lesson, we will mostly work with the `"executionStats"` outputs a series of objects containing detailed information about our operation. We won't include the entire output below, but rather we'll focus on a specific object, called `executionStats`.

If we were to execute the `.explain()` method before creating our index on the `year_abroad` field, the output might look something like this:

```
executionStats: {
  executionSuccess: true,
  nReturned: 1336,
  executionTimeMillis: 140,
  totalKeysExamined: 0,
  totalDocsExamined: 5555,
  executionStages: {
    …
  }
}
```

Examine the To see `nReturned`, `totalDocsExamined`, and `executionTimeMillis` fields. Notice that out of 5555 total documents, only 1336 were returned by our query, which took approximately 140 milliseconds.

Now let's look at what the output of our query might look like after we index the `year_abroad` field:

```
executionStats: {
  executionSuccess: true,
  nReturned: 1336,
  executionTimeMillis: 107,
  totalKeysExamined: 1336,
  totalDocsExamined: 1336,
  executionStages: {
   …
  }
}
```

```
```

Check out the `nReturned` and `totalDocsExamined` fields again. Notice anything?

When we ran our query The above command would create an index on all the documents in the student's collection based on the *after* creating our index, we still returned 1336 documents, but instead of examining the entire collection, 5555 documents, we only examined the 1336 we returned. This is because our query first scanned the index to identify documents that matched our filter, then returned only the corresponding documents without browsing every document in the collection.

Take a look at the We can run a query on the indexed field to utilize the indexed `executionTimeMillis` for each query. You'll also notice that our query after creating the index took 107 milliseconds, while our query before creating the index took a bit longer, 140 milliseconds. This might not seem like much, but if we were working with a collection containing tens or hundreds of thousands of documents, the time difference would likely be much more significant.

Compound Indexes

In addition to single field indexes, MongoDB gives us the ability to create compound indexes. In MongoDB, indexes play an important role in making sure our Compound indexes contain references to multiple fields within a document and support queries that match on multiple fields. Let's have a look at the syntax for creating a compound index:

```
db.<collection>.createIndex({
  <field>: <type>,
  <field2>: <type>,
  …
})
```

Similar to single field indexes, MongoDB will scan our Note that the method is appended to the index for matching values, then return the corresponding documents. With compound indexes, the order of fields is important. To understand why, let's return to our example from the first exercise about the university president.

Imagine that as president we wanted to plan reunions not just for students who studied abroad during a particular year, but also for students who studied abroad in a particular country. We could create a compound index on two fields: To see `study_abroad_nation` and `year_abroad`, like so:

```
db.students.createIndex({
  study_abroad_nation: 1,
  year_abroad: -1
});
```

This creates a single index that references two fields: In this lesson, we will mostly work with the `study_abroad_nation` in ascending, or alphabetical order, and `year_abroad` in descending, or reverse chronological order.

Because of how the fields are ordered, references within this index will be sorted first by the If we were to execute the `study_abroad_nation` field. Within each value of the `study_abroad_nation` field, references will be sorted by the `year_abroad` field. This is an important consideration in determining how well our indexes will be able to support sort operations in our queries.

Now that we've created this compound index, anytime we query on these two fields, MongoDB will automatically employ this index to support our search.

The below query would be a use case for our compound index:

```
db.students.find({
  study_abroad_nation: "Brazil",
  year_abroad: 2012
});
```

Compound indexes can also support queries on any **prefix**, or a beginning subset of the indexed fields. For example, consider the following compound index:

```
db.students.createIndex({
  study_abroad_nation: 1,
  year_abroad: -1,
  graduation_year: 1
});
```

In addition to supporting a query that matches on the When we ran our query The above command would create an index on all the documents in the student's collection based on the `study_abroad_nation`, `year_abroad` and `graduation_year` fields, this index would also be able to support queries on the following fields:

- `study_abroad_nation`
- `study_abroad_nation` and `year_abroad`

This index would *not*, however, be able to support queries on the following fields:

- `year_abroad`
- `graduation_year`
- `year_abroad` and `graduation_year`

As each index must be updated as documents change, unnecessary indexes can affect the write speed to our database. Make sure to consider if a compound index would be more efficient than creating multiple distinct single-field indexes to support your queries.

Multikey Index on Single Fields

How do MongoDB indexes handle fields whose values are arrays? Conveniently, MongoDB automatically creates what's known as a In MongoDB, indexes play an important role in making sure our **multikey index** whenever an index on a array field is created. Multikey indexes provide an index key for each element in the indexed array.

Suppose we had a document within the `students` collection that had a field, `sports` with an array as its value:

```
{
  _id: ObjectId(...),
  last_name: "Tapia",
  first_name: "Joseph",
  major: "architecture",
  birth_year: 1988,
  graduation_year: 2012 ,
  sports: ["rowing", "boxing"]
}
```

We could create a multikey index on this field in the same way we would create any other single-field index:

```
db.students.createIndex({ sports : 1 });
```

This would create an index that references the In this lesson, we will mostly work with the `sports` field for every document in the collection. Since `sports` is an array field, the resulting multikey index would contain individual references to each element in the array. We specified ascending order for our index so the values would be organized in alphabetical order.

Note that this example discusses multikey single field indexes. Next we'll learn about some important considerations to keep in mind when creating compound multikey indexes.

Multikey Index on Compound Fields

Is it possible to create a compound multikey In MongoDB, indexes play an important role in making sure our index in MongoDB? The answer is yes, with a very important caveat - only *one* of the indexed fields can have an array as its value.

For example, suppose we had a document within a Suppose we had a document within the `students` collection with two fields with arrays as their values: `sports` and `clubs`.

```
{
  _id: ObjectId(...),
  last_name: "Tapia",
  first_name: "Joseph",
  major: "architecture",
  birth_year: 1988,
  graduation_year: 2012 ,
  sports: ["rowing", "boxing"],
  clubs: [“Honor Society”, “Student Government”, “Yearbook Committee”]
}
```

A single compound index can not be created on *both* the `sports` and `clubs` fields. We could, however, successfully create a compound multikey index on `sports` *or* `clubs` along with any of the other fields.

For example, either of the following would successfully create a compound multikey index:

```
db.students.createIndex({ sports: 1, major: 1 });
db.students.createIndex({ clubs: -1, graduation_year: -1 });
```

If we wanted to index both the `sports` and `clubs` fields, we'd have to create two separate indexes for them.

Deleting an Index

Each time we make changes to a collection, any indexes associated with that collection must also be updated. In this way, unnecessary indexes can slow down the performance of certain CRUD operations. This is why it is important to review our indexes

and remove any that are redundant or not being used.

Suppose, after some reflection, we discovered that a compound Suppose we had a document within the index can handle all the queries we need to make, instead of the single field indexes we originally were relying on. Once we've created the compound index, it would be a good idea to identify and remove any unnecessary indexes.

First, we can use the `.getIndexes()` method to see all of the indexes that exist for a particular collection.

Consider a collection called A single compound index can not be created on `students` that has multiple indexes:

```
db.students.getIndexes();
```

Might output:

```
[
   {  v : 1,
      key : { _id : 1 },
      name : '_id_'
   },
   {
      v : 1,
      key : { sports : -1 },
      name : 'sports_-1'
   },
   {
      v : 1,
      key : { sports : -1, graduation year : -1 },
      name : 'sports_-1_graduation_year_-1'
   }
]
```

Now that we have a list of our indexes for the `students` collection, we can see that both the second and third indexes index the `sports` key in descending order. Since compound indexes can support queries on any of the prefixed fields, our third index, named `'sports_-1_graduation_year_-1'`, can support queries on both `sports` and `graduation_year`.

This means that our second index, `'sports_-1'`, is redundant. MongoDB gives us another method, `.dropIndex()`, that allows us to remove an index, without modifying the original collection. We can use it to delete the `'sports_-1'` index:

```
db.students.dropIndex('sports_-1');
```

The above command would delete the index, and then we can confirm by running `db.students.getIndexes();` again:

```
[
   {  v : 1,
      key : { _id : 1 },
      name : '_id_'
   },
   {
      v : 1,
      key : { sports : -1, graduation year : -1 },
```

```
      name : 'sports_-1_graduation_year_-1'
   }
]
```

Getting rid of unnecessary indexes can free up disk space and speed up the performance of write operations, so as you start to use indexes more, it is worth regularly scrutinizing them to see which, if any, you can remove.

Takeaway

- An index is a data structure that captures a subset of a collection's data in an easy-to-traverse form. We can use the `.createIndex()` method to create an index.
- A single field index is an index that references one field from a document.
- We can use the `.explain()` method with the `"executionStats"` argument to gain insight into the performance implications of our index on our query.
- A compound index is an index that contains references to multiple fields within a document.
- Multikey indexes are automatically created whenever we create an index on a field that contains an array value. Multikey indexes create an index key for each element in the array.
- A compound index cannot support two multikey indexes.
- The `.dropIndex()` method deletes an index without modifying the original collection.

In addition to the syntax we've learned throughout this lesson, MongoDB offers us other syntax and commands that can be useful when indexing collections:

- Partial Indexes only index documents in a collection that meet specific filter criteria. By indexing a subset of a collection's documents, partial indexes consume less storage and have improved performance.
- Sparse Indexes only index documents that include the specified index field. Any documents that do not have the field will be excluded from the index. Much like partial indexes, these indexes can use significantly less storage and have relatively improved performance compared to non-sparse indexes.
- TTL Indexes are special single-field indexes that MongoDB can use to automatically remove documents from a collection after a certain amount of time or at a specific clock time.
- Unique Indexes enforce unique values for the indexed fields. Creating a unique index on a collection will restrict the insertion or update of documents where the indexed field's value matches an existing value in the index.

Explore MongoDB Aggregation

Learn what aggregation

Learn what aggregation is and how to use it in MongoDB!

Introduction

When working with MongoDB, we will likely need to perform various operations on our data. At some point, we may decide that we want to perform some form of analytics. Take, for example, a database that stores sales data for an online store. We can use the

methods we have learned so far (e.g., `find()`) to perform CRUD operations related to the data, but maybe we want to answer questions related to trends like:

- What kind of products are selling the best over a six-month period?
- What product is making the most sales on Wednesdays?
- Is there a specific geographic location that tends to order more products than others?

These questions are best answered by performing more complex analytical operations on our database. With some databases, we might have to use a completely separate tool set to perform analytics. However, with MongoDB, the ability to perform these types of queries is already built-in! This process is known as **aggregation** and is one of the core features of a MongoDB database. In this article, we will explore aggregation and how it works in MongoDB.

Specifically, we will:

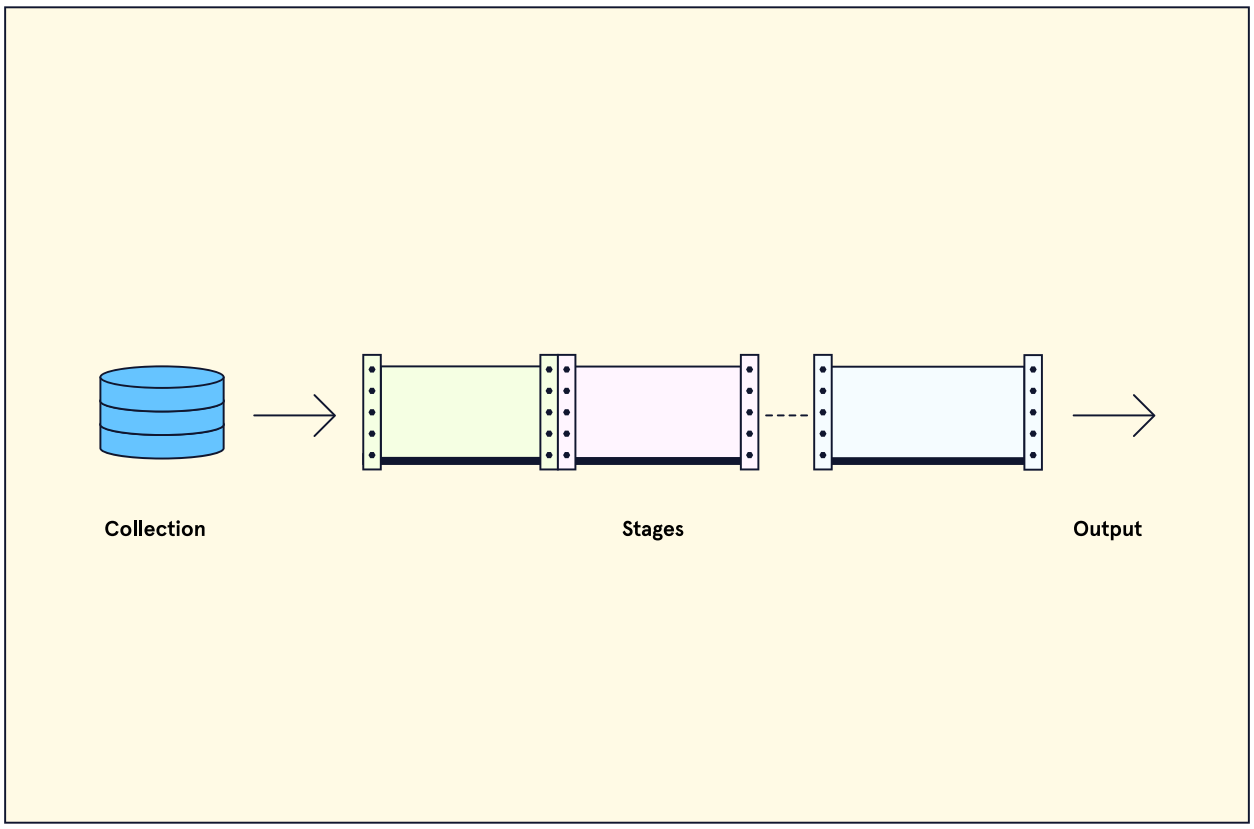
- Explore the basics of how the aggregation pipeline works in MongoDB.
- Build an aggregation pipeline together using multiple stages and operators.
- Consider the right situations to use aggregation.

Let's get started!

Aggregation Basics

In plain terms, to aggregate means to combine out of several parts. When we apply this concept to a MongoDB database, aggregation is the process by which we can sift through large amounts of data one step at a time and, at each step, perform some form of filtering or computation on the data. Then, after multiple steps, we return a final result. This process can help us to see our data in a new way and provide valuable insights. So, how do we actually perform aggregation? One of the primary ways to accomplish aggregation in MongoDB is to use an **aggregation pipeline**.

An aggregation pipeline is a channel through which data passes from point A (the start of the pipeline) to point B (the end of the pipeline). Imagine, though, that the pipe is split into a number of segments. Each of these segments in the aggregation pipeline is called a **stage**, and each stage performs a specific operation on the data, such as sorting or filtering. Take a look at the following visual to help paint a picture of the pipeline:



If we use the above image as a guide, we can note that at the start of our pipeline, we will have our original dataset (a collection). Then, at the first stage and at successive stages, an operation is performed on the data, and the result is either sent to the next stage or returned if there are no more stages. There can be many stages involved depending on what we might be trying to accomplish with our pipeline.

Now that we know what aggregation and an aggregation pipeline is, let's see how we can create our own in MongoDB.

Getting Started with Aggregation

To start using aggregation via an aggregation pipeline in MongoDB, we can use the following `.aggregate()` method like so:

```
db.<collection>.aggregate()
```

MongoDB requires that inside of the `aggregate()` method, our first argument is an array containing the pipeline stages we use. To see a pipeline stage in action, let's imagine we had the following small collection called `movies`, with each record having a field with the movie name and an associated rating (using the USA-based MPA rating system):

```
{
  name: "Star Wars: Clone Wars"
  rating: "PG"
},
{
  name: "Indiana Jones and the Temple of Doom"
  rating: "PG"
},
{
```

```
  name: "Despicable Me"
  rating: "PG"
},
{
  name: "The Godfather"
  rating: "R"
}
```

We saw earlier, that in order to build a pipeline, we will need to define the stages we want to use. There are many stages in MongoDB that help accomplish various tasks in aggregation. For now, let's use a common stage called `$match` that returns all the documents containing the specified field and value. This is similar to when we used the `find()` method and provided a query argument to filter a document based on a specific criteria.

The syntax to accomplish this aggregation would be the following:

```
db.movies.aggregate([
  {
    $match: {rating: "R"}
  }
])
```

And our result would be:

```
{
  name: "The Godfather",
  rating: "R"
}
```

Building stages into the pipeline will be the bread and butter of working with aggregation. Now that we have the basics down and have seen how we can start to build a pipeline, let's start to walk through how to build a more complex multi-stage pipeline. Along the way, we will explore new stages and deepen our understanding of aggregation!

Aggregation in Action: Building a Multi-Stage Aggregation Pipeline

Imagine we had a large data set (thousands of records) of students in a school. Each student document has the following structure:

```
{
  student_id: 94204,
  first_name: "Sun",
  last_name: "Ko",
  grade_level: 6,
  test_scores: [99, 97. 96, 99],
  average_test_score: 98.65
}
```

We are tasked with gaining insights on students that might qualify for a prestigious scholarship. We need to produce a *new* collection in our database called `candidates` that has the following criteria:

- The collection must only contain students in the 6th grade with an average test score above 97.
- The students in the new collection must be sorted by their first name (`first_name` field).

We also want the records in the new collection to have a new field called `highest_score` with the value of the highest score in the `test_scores` array field. This type of task would be a great candidate for using the aggregation framework!

We can start our aggregation pipeline with a first stage that filters out only students in the 6th grade with an average test score above 97. Just like before, we can use the `$match` stage:

```
db.students.aggregate([
  {
    // First stage
    $match: {grade_level: 6, average_test_score: {$gt: 97}}
  }
])
```

This first stage with `$match` should give us all the students that meet the conditions to qualify.

Next, we want to sort our result. Thankfully, MongoDB provides a stage named `$sort`. Similar to how the `.sort()` method works, we can specify -1 or 1 to sort in ascending or descending order for a field. Here is what our pipeline would look like if we added a sorting stage:

```
db.students.aggregate([
  {
    // First stage
    $match: {grade_level: 6, average_test_score: {$gt: 97}}
  },
  {
    // Second Stage
    $sort: { first_name: 1}
  }
])
```

This new stage would take the resulting collection from the first stage (where we used `$match`) and sort the documents in ascending order by the `first_name` field.

At this point, we might think, “Why use an aggregation pipeline when I can accomplish the same goal with `find()` and `sort()` much quicker?”. That is a valid point, and if you likely had to just do those two stages, using `find()` and `sort()` likely would be quicker. However, the next two stages are where aggregation would start to shine as a better alternative.

First, let’s accomplish adding a new field to our records called `highest_score` with the value of the student’s highest test score. We can do so using the `$addFields` stage. Before we add it to our pipeline, let’s examine the syntax of this operator:

```
{ $addFields: { <newField>: <expression>, …}}
```

Notice that this stage uses what is known as a **expression**. Aggregation expressions are commonly used in stages to perform some type of logic such as arithmetic or comparisons. There are many types of expressions including: literals, system variables, expression objects, and expression operators. Here is what our pipeline looks like if we needed to get the highest score and create a new field:

```
db.students.aggregate([
  {
    // First stage
    $match: {grade_level: 6, average_test_score: {$gt: 97}}
  },
  {
    // Second Stage
    $sort: { first_name: 1}
  },
  {
    // Third Stage
    $addFields:  {
      highest_score: { $max: "$test_scores" }
    }
  }
])
```

Here, we are using the `$addFields` stage with the `$max` expression operator, a specific type of expression, which allows us to pull the max value of the `test_scores` field so we can use it in our new `highest_score` field. Note that our `test_scores` field is prefixed with a `$` to indicate it is a **field path**. Field paths are used to access a document's fields inside of an expression. We will have to use field paths often when working with aggregation. In this case, it allows us to access the `test_scores` field from our documents to use with the `$max` expression operator.

If we left it as is, our result from our aggregation pipeline would have documents like this:

```
{
  student_id: 94204,
  first_name: " Sun",
  last_name: "Ko",
  grade_level: 5,
  test_scores: [99, 97. 96, 99],
  average_test_score: 87.5,
  highest_score: 99
}
```

So far, so good! We just have one final task in our pipeline: creating a new collection. Creating a new collection can be accomplished by using the `$out` stage. Here's how our pipeline would look after adding the `$out` stage:

```
db.students.aggregate([
  {
    // First stage
    $match: {grade_level: 6, average_test_score: {$gt: 97}}
  },
  {
    // Second Stage
    $sort: { first_name: 1}
  },
  {
    // Third Stage
    $addFields:  {
```

```
      highest_score: { $max: "$test_scores" }
    }
  },
  {
    // Fourth Stage
    $out : "candidates"
  }
])
```

The `$out` stage can output the final result of an aggregation pipeline to a new database, a new collection, or both! For this reason, it is required that it is the last stage in a pipeline. In this case, our aggregation result would be plopped into a new collection named `candidates`.

Whew! We did it. We went through a general overview of how to build a multistage pipeline, step by step, with a few common aggregation stages. Note that the order of the aggregation stages is important since data will flow in the order we supply our operators.

Now that we have had a chance to build a pipeline, it's worth thinking about when we actually need to use aggregation and build our own pipelines.

When to Use Aggregation

When we compare the ways we can manipulate data with methods like `find()` and `updateOne()` to an aggregation pipeline, we can start to see a major difference.

Most of the CRUD methods that MongoDB offers are operational in nature. Their role is to perform some specific operation on our data and that's it. With aggregation pipelines, we are able to perform multiple operations together to curate data that is more analytical in nature. This helps us see our data in a bigger picture.

A good way to picture the difference is to return to a web application that makes sales of some product. The application needs to give users the ability to perform common CRUD operations like adding a product to their cart or increasing the quantity they want to purchase. These types of operations are best handled with the functions we already know (e.g, `find()`, `updateOne()`). However, if we had an option to have a seller dashboard that can give insights on product sales or trends, our CRUD methods wouldn't be as helpful since they're not able to manipulate any of the data. That's where we can see aggregation really shine!

In essence, consider using aggregation when:

- There are no CRUD methods (or a combination of methods) that accomplishes the query that needs to be performed easily.
- We need to perform analysis on datasets such as grouping values from multiple documents, computations on data, and analyzing data changes over time.

Wrap Up

In this article, we learned about what makes aggregation a powerful tool for searching and filtering data in MongoDB. Let's take a moment to review what we've learned:

- In MongoDB, we can perform aggregation as an alternative way to query data.
- One way of accomplishing aggregation is by using an aggregation pipeline via the `.aggregate()` method.
- Aggregation pipelines allow us to incrementally filter data through the use of stages, where each stage filters/modifies the data in a specific way and then passes that data to the next stage.
- We can build a pipeline using stages such as `$match` or `$sort`.
- Some stages can utilize different types of expressions such as expression operators like `$max`.
- To reference fields from the documents in our collections inside of expressions, we must use a field path.

- Aggregation is particularly useful when we have tasks that can't be accomplished with common CRUD methods easily or when we are looking to perform complex analytics on datasets.

Using the aggregation framework will open new doors to how we can query and analyze large datasets. To learn more about aggregation in MongoDB, check out these helpful resources:

- MongoDB: Aggregation Basics
- MongoDB: Aggregation Pipeline Introduction
- MongoDB University: M121 Aggregation Course

Explore MongoDB Atlas

Learn about MongoDB Atlas and how to get started using it!

Introduction

We've decided to start a web-based business that sells used cameras and lenses called "Lenses for Less". Understanding the power and versatility of MongoDB, we want to use MongoDB to store the data related to purchases, inventory, and all the other data necessary for your new business. To store our data, we can purchase or rent a physical server (aka a powerful computer) and run an instance of MongoDB on it – occasionally upgrading or adding new servers as our business scales. This would be a large undertaking since we wouldn't only have to spend money on a server, but we would also have to maintain it over a long period of time. Nowadays, it makes more sense to use a cloud offering instead of building from scratch. Thankfully, MongoDB Atlas offers a fully managed solution that allows us to get up and running in minutes without worrying about the complexities of managing our own software infrastructure.

In this article, we will learn about what the MongoDB Atlas platform is and how it works. Specifically, we will:

- Examine the Atlas platform and the services it offers.
- Dive into how MongoDB stores our data in the cloud.
- Deploy our own MongoDB Atlas Custer and connect to it locally.

What is MongoDB Atlas?

MongoDB Atlas is a developer data platform. It includes a suite of cloud databases and data services. For the purposes of working with databases, Atlas hosts a variety of features that help us quickly set up, deploy, and maintain a MongoDB database. Atlas allows us to store and manage our data in the cloud through an easy-to-use website interface. With Atlas, we can have a MongoDB database set up and running in just a few clicks!

On top of simply storing our data, Atlas offers several different integrated features to help us make the most of our data. A few of these are:

- Atlas Search - which allows for quick and easy text-based queries of data stored in the cloud.
- Atlas Charts - provides data visualization, which is fully integrated with the data we store with Atlas.
- Atlas Data Lake - helps perform large-scale analytics on our data.

We get all this and more right out of the box with Atlas. This makes it a great solution for not just small businesses like "Lenses for Less" but also major corporations like Verizon or Toyota. Before we jump into how to set up our own Atlas instance, let's learn about how Atlas stores data!

Atlas Data Storage

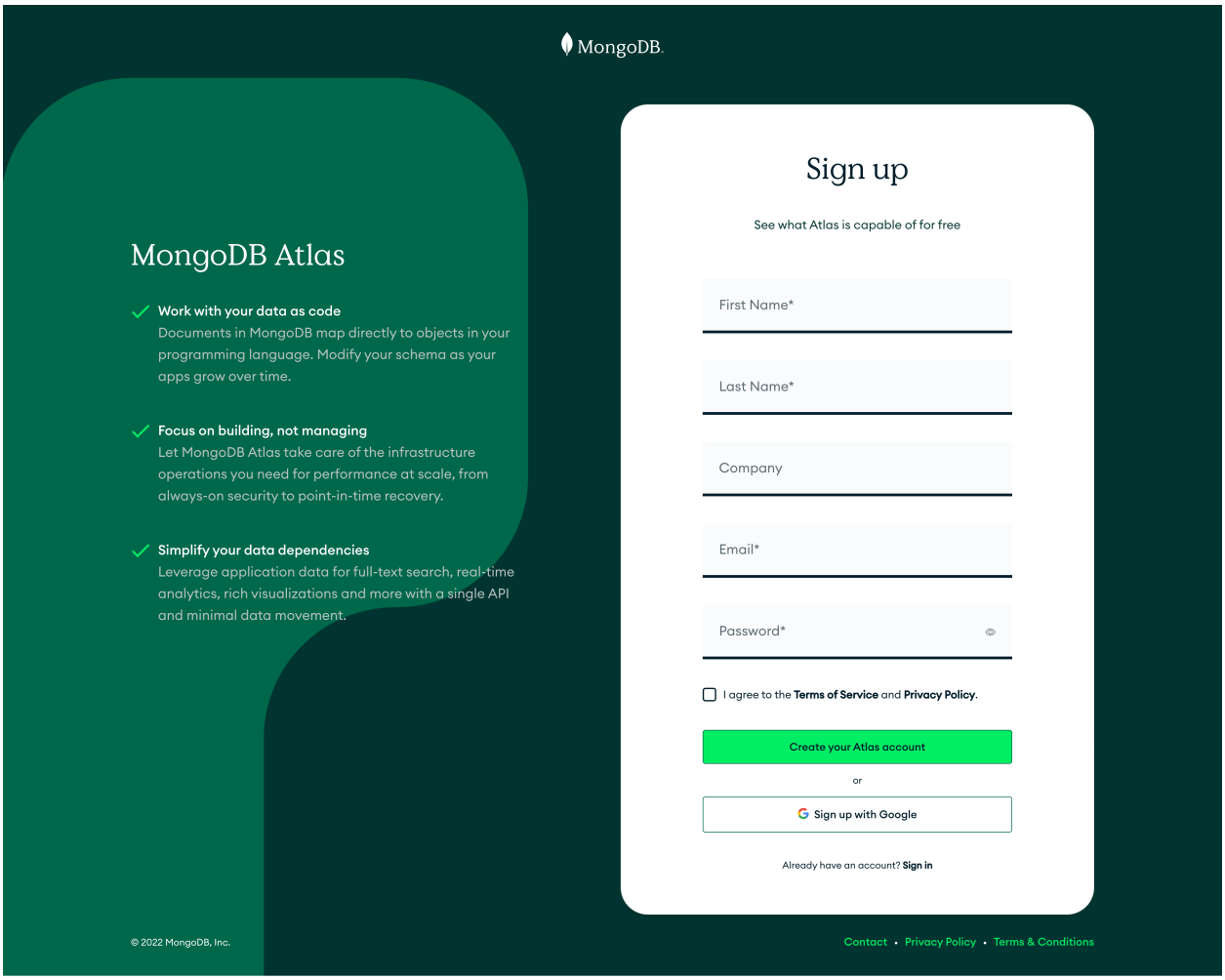
In Atlas, we interact with our data in what is known as a **cluster**. We can think of clusters as a unit of storage that MongoDB uses to house data. Depending on the plan we choose for our account, we can end up using clusters in a slightly different way. Atlas offers three different plan types:

- Free: Atlas offers a free plan that allows users to get started quickly without any worry of payments or budget. The free plan comes with 0.5GB of free storage and a set of basic configuration options. This plan is great for learning and exploring MongoDB in a cloud environment.
- Serverless: This plan is Atlas' serverless database offering, which means users can create a database for their application without having to worry about security, reliability, managing performance, or managing scale. Serverless offers operations-based pricing that charges based on reads, writes, and storage. It's a great option for applications that might have sparse or infrequent traffic.
- Dedicated: This plan is Atlas' dedicated multi-region cluster offering. Dedicated clusters can be customized and optimized for specific workload requirements (e.g. higher CPU speeds and more memory), and have predictable pricing. Advanced security and multi-region options make this a great option for individuals and businesses running critical applications.

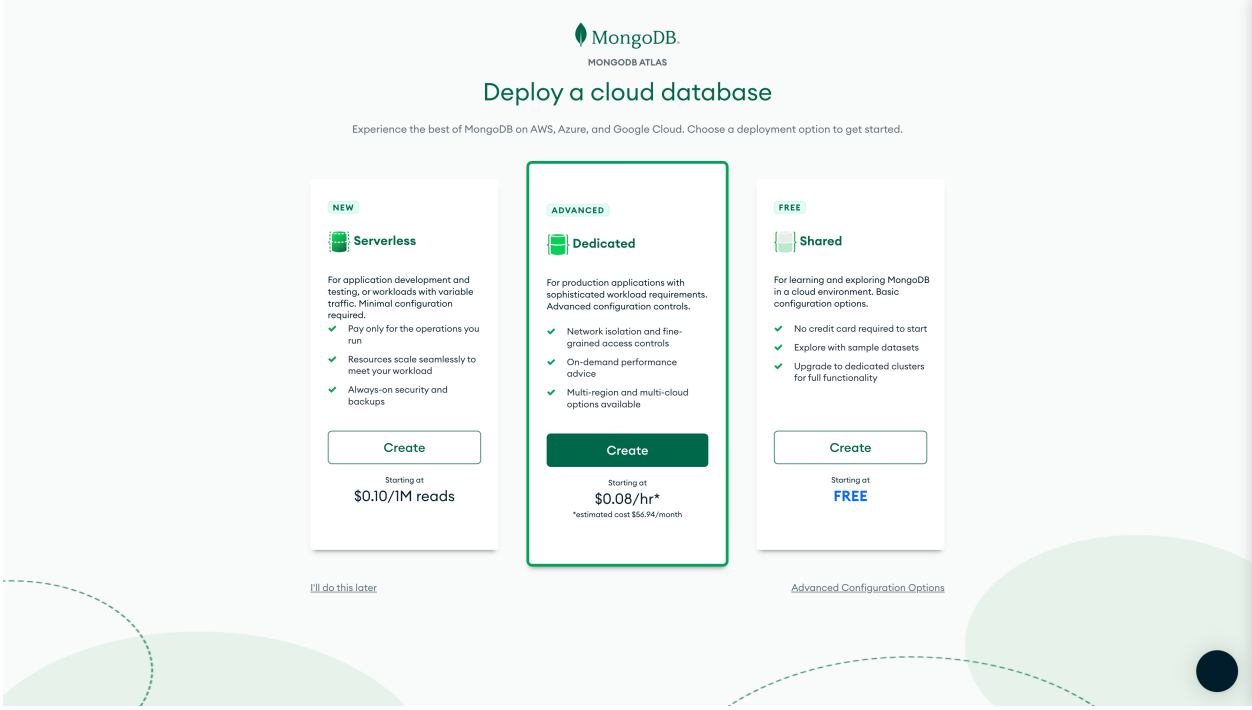
For the purposes of this article, we will be using the free plan. Now that we know how Atlas stores our data, let's go through the basic setup of a free Atlas cluster!

Setting Up an Atlas Deployment

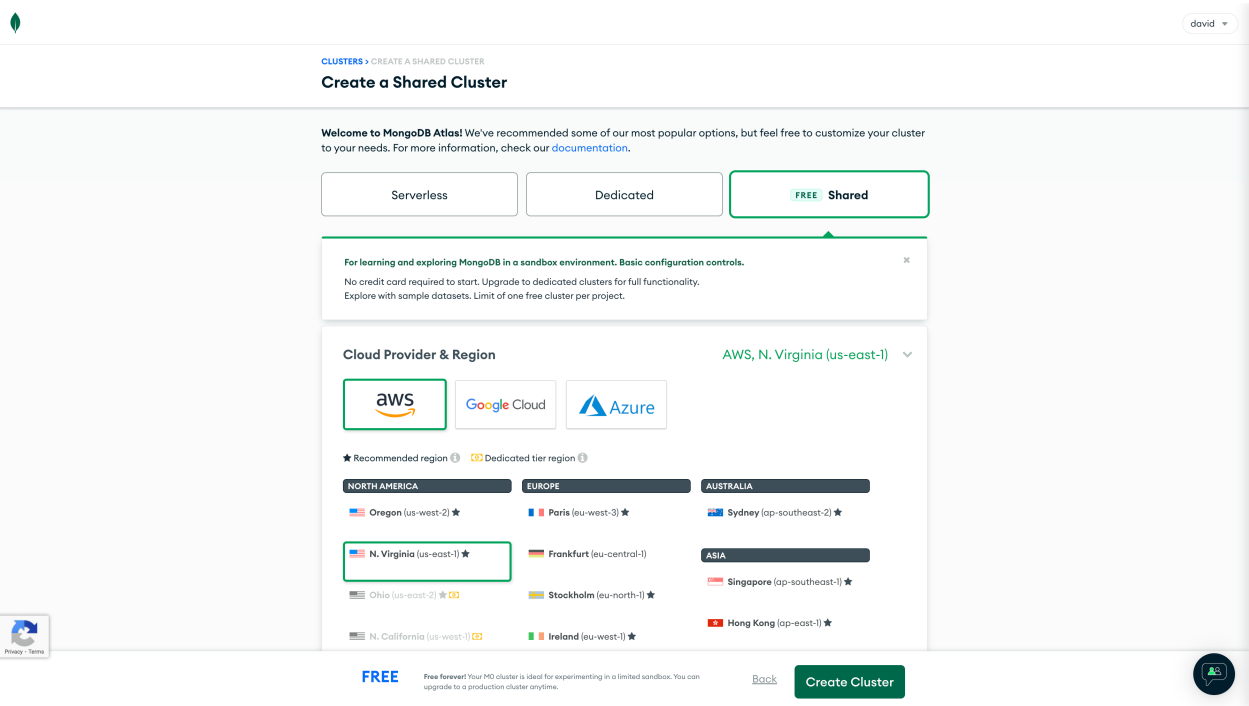
To start setting up our own Atlas cluster, we will need to register for an account. Get started by visiting the MongoDB MongoDB registration page page and signing up for an account:



Now that we have an account, after a few more minor steps (e.g., privacy policy, onboarding form), we should be directed to our deployment setup page. For the purposes of this article, we can use the free plan. It should look like this:



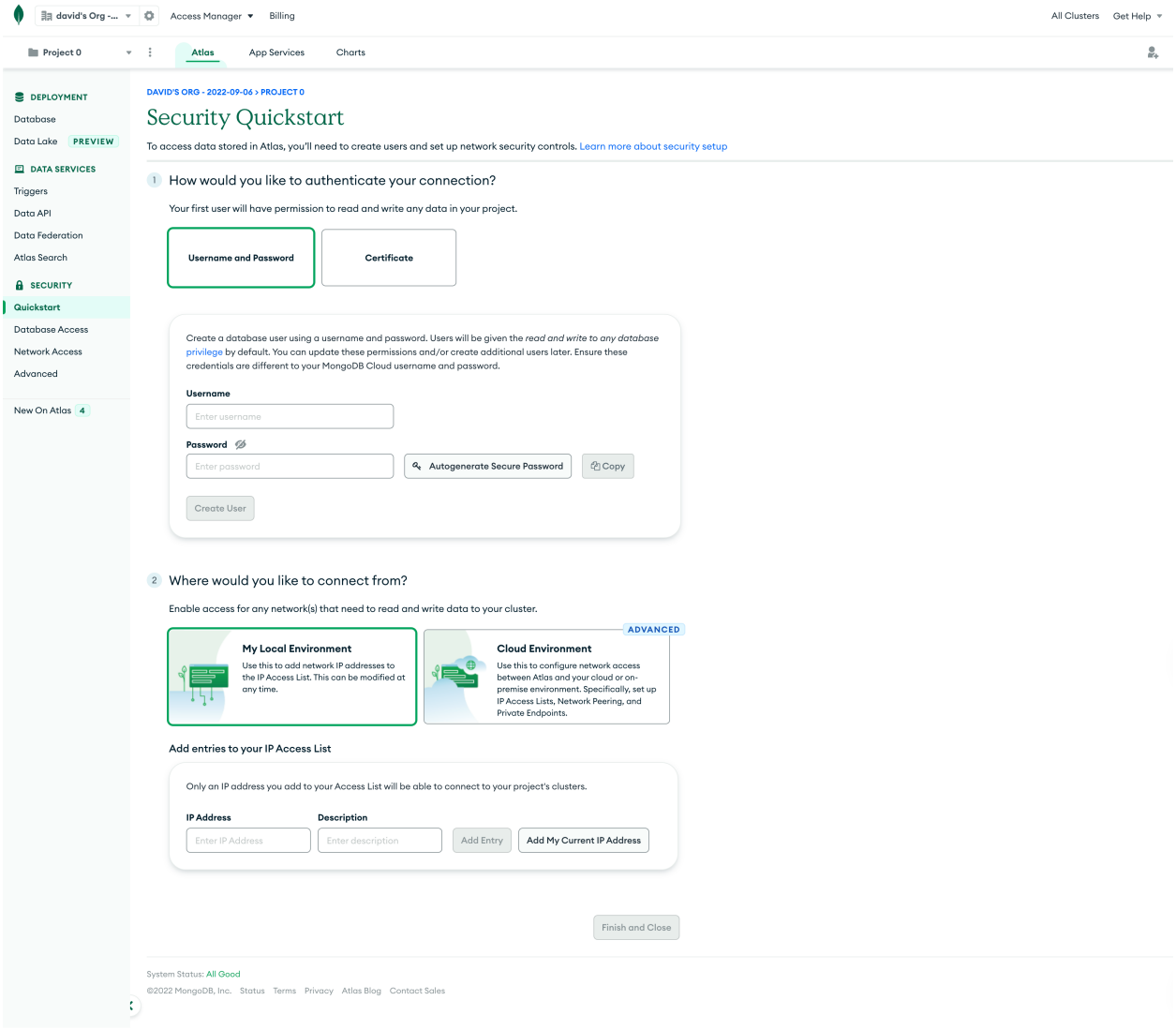
Then, we can select some settings for our cluster:



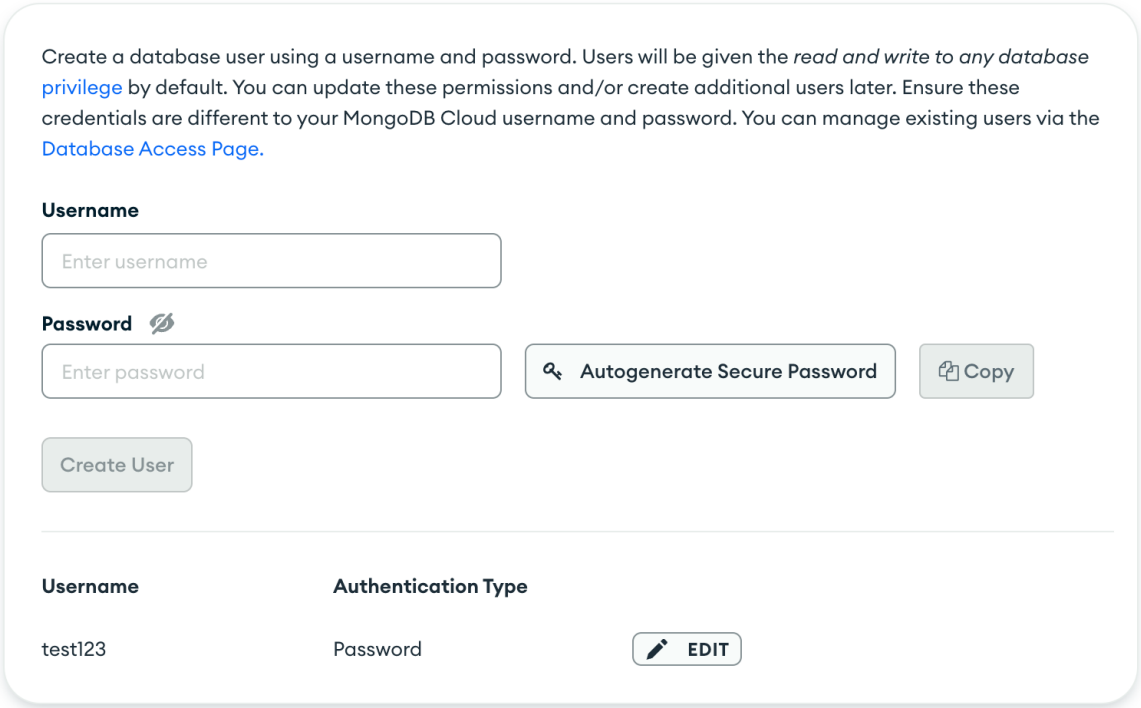
We can leave the settings as their default. Note that Atlas automatically picks the closest regions for you, depending on your location. While we are using the default settings here, it's worth noting that in the future, we will have the choice to host our database on a variety of cloud platforms such as AWS, Azure, Google Cloud, or several other options.

Click the "Create Cluster" button on the right side of the bottom of the page to choose a cluster. MongoDB will begin building a cluster for us (there will be a small notification on the bottom left side of the screen showing its progress) and redirect us to a

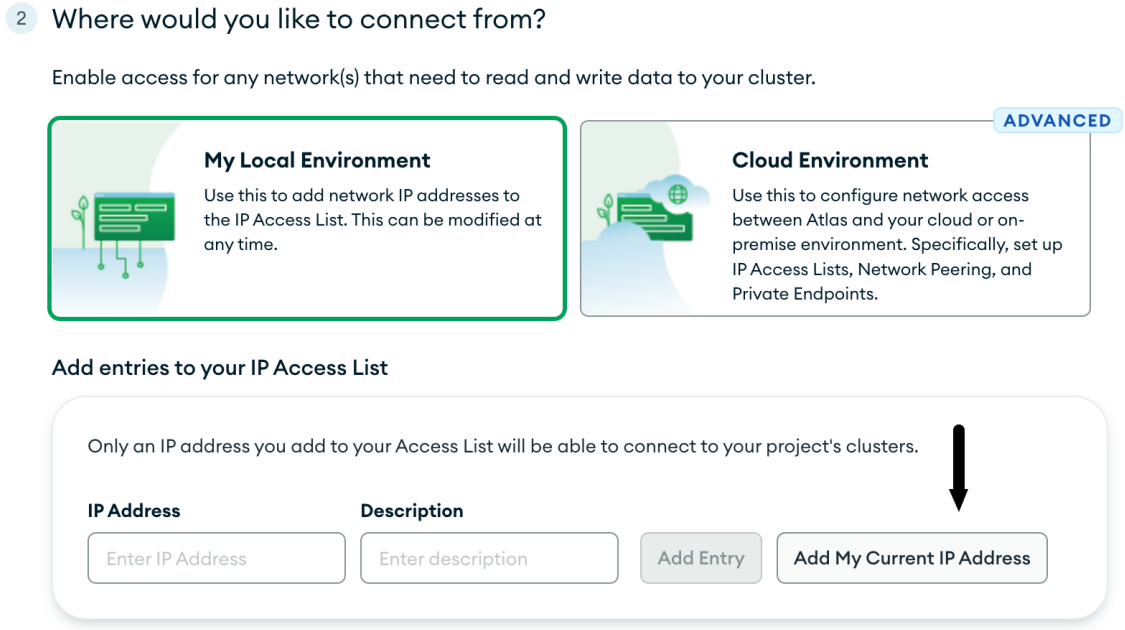
security setup page:



For us to access our MongoDB cluster, we will need to set up proper access. On this security page, set up a user by entering a username and password. We can also have MongoDB autogenerate a password for us. Click the “Create User” to add the user to our database. When we are finished, it should look like this (note the user “test123”):



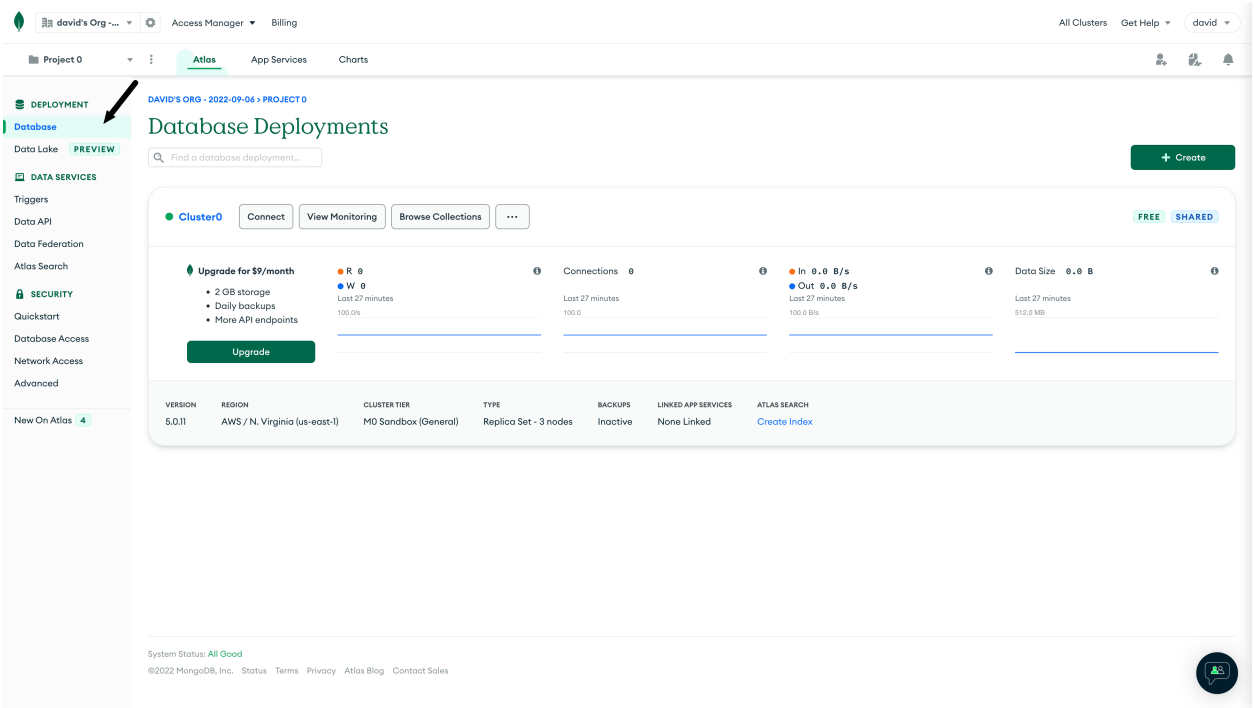
Next, we will need to enable access for connecting to the MongoDB database via our computer. We can do so by adding our IP address to the Atlas security settings to allow us to access our cluster from our computer. Click the “Add My Current IP Address” button:



MongoDB will pull our current IP address and add it to an IP access list. That wraps up our initial setup. Now, the fun part, accessing our database!

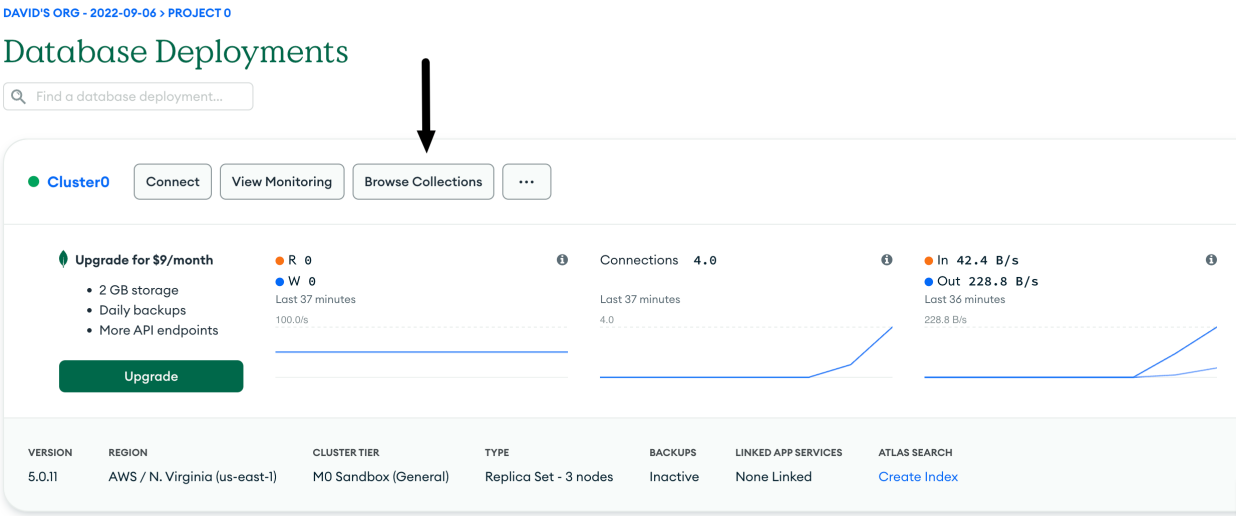
Set Up an Atlas Database

It's time to start navigating and working with our Atlas cluster. Let's start by navigating to our database dashboard. Click the "Database" tab under the "Deployment" header on the left-hand side:

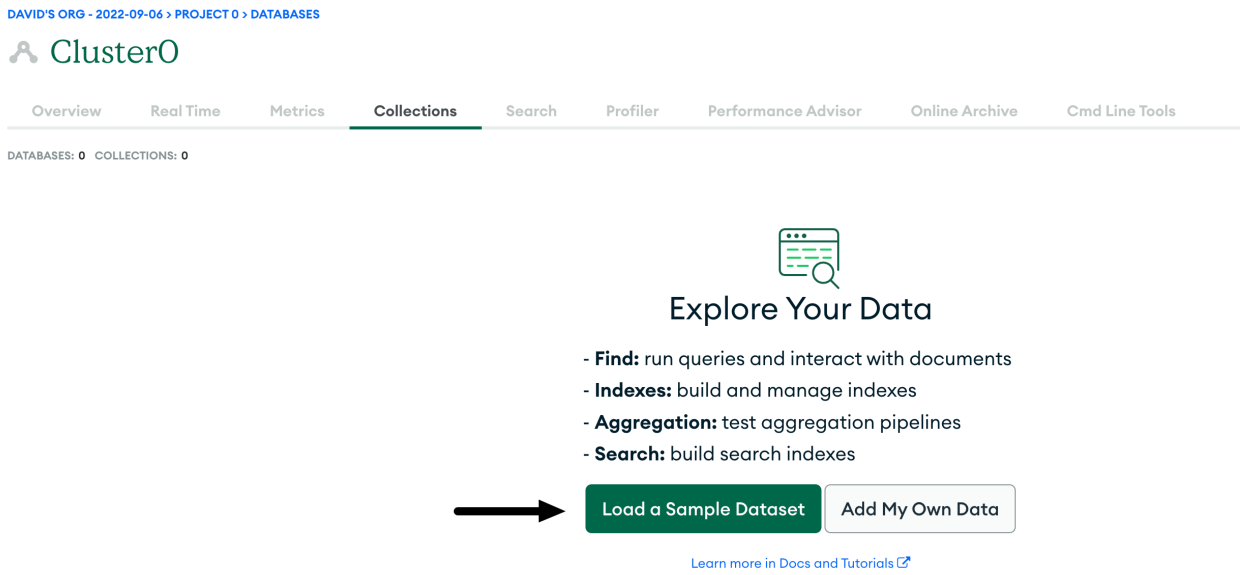


This dashboard is a central location for managing the database component of our cluster. This dashboard shows important information about our database, such as its size and connections.

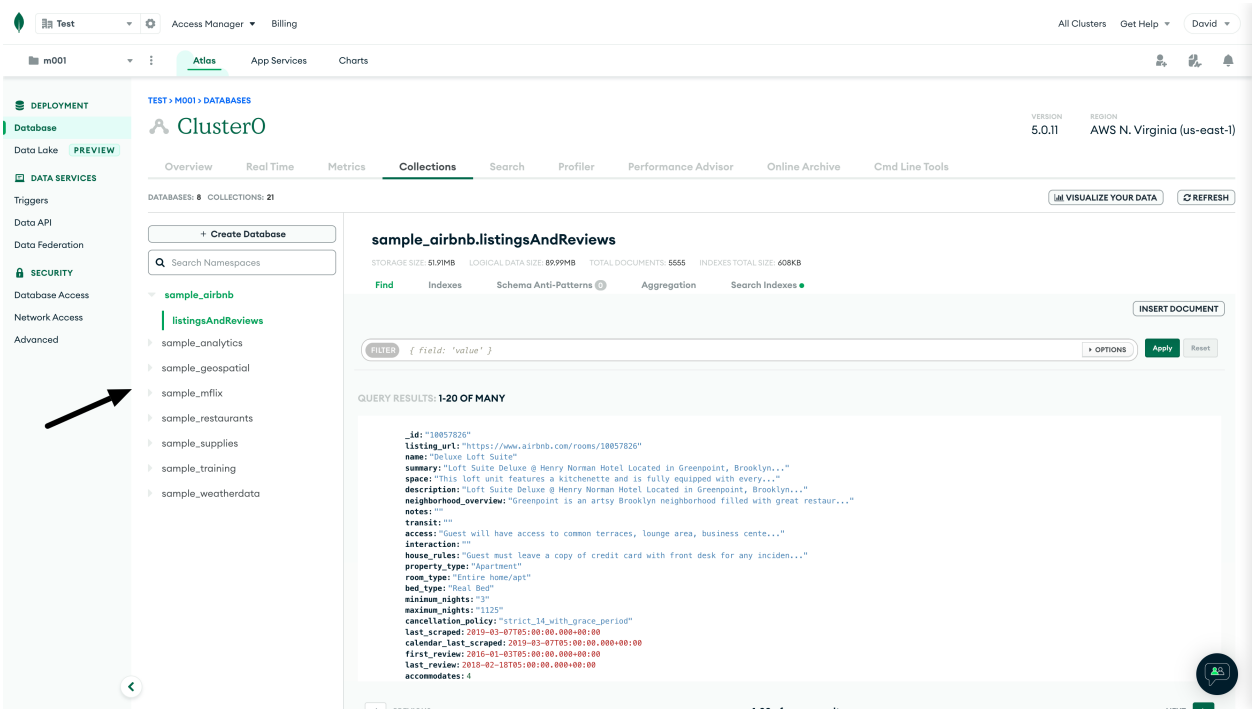
Our database will start off empty, but MongoDB allows us to fill it with a sample dataset (or even our own data). To do so, click the "Browse Collection" button on the database cluster section:



From here, we will be able to load a sample dataset by clicking the "Load Sample Dataset" button:



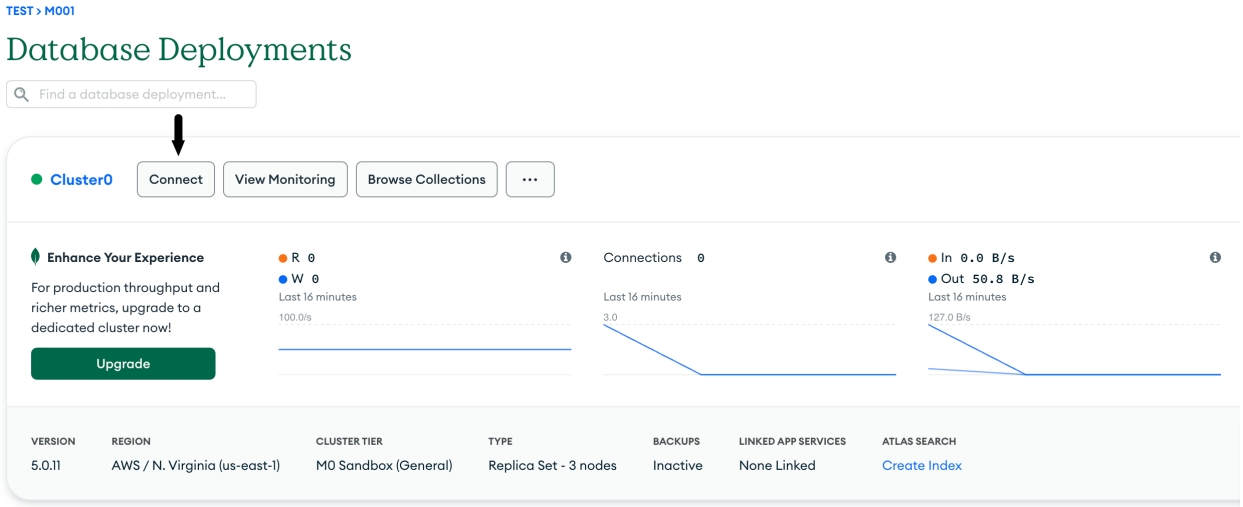
It may take several minutes for the data to load, but it will exist inside the cluster once the process is complete. Notice that there is a number of databases that MongoDB creates for us:



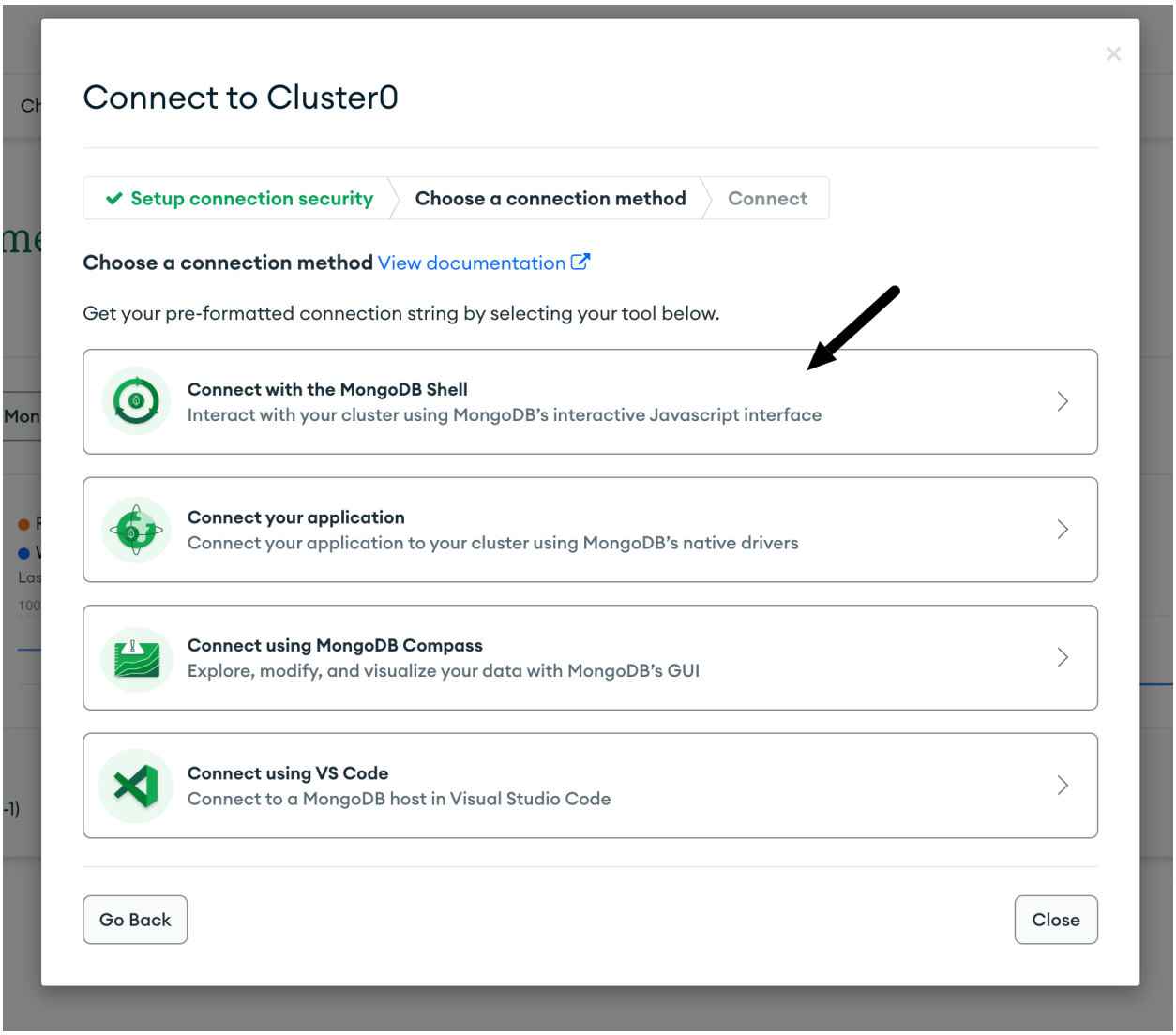
From our dashboard, we can browse our databases and their respective collections and query those collections for specific data. There are a number of other features the dashboard offers. We won't cover them in this article, but we encourage you to explore!

Connecting to Atlas

Now that we have an Atlas cluster setup with some databases, let's see how we can connect to it via our local machine (computer). Navigate back to the main dashboard for the database by clicking the database tab on the left-hand side again. From here, we have the option to connect to the Atlas cluster by clicking the "Connect" button:

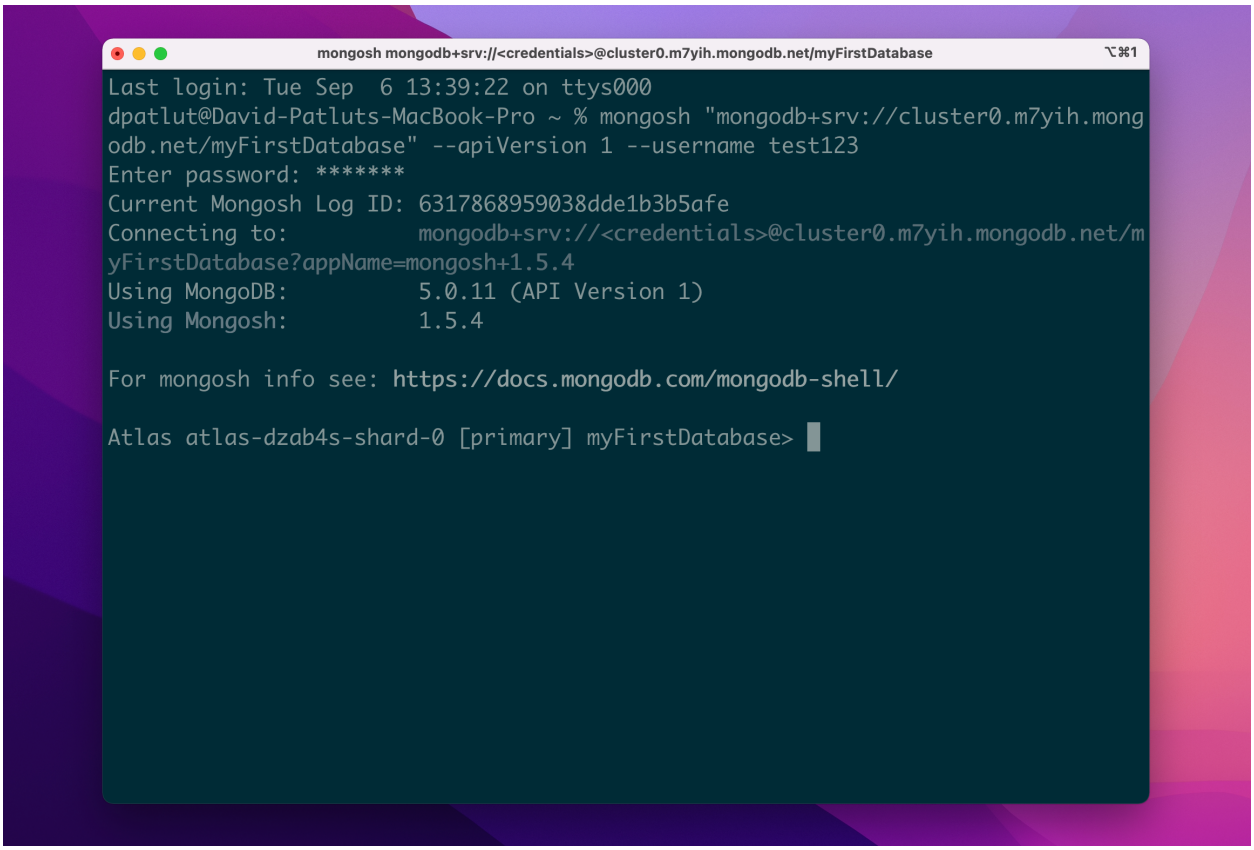


The menu for the connection offers us several options to connect to Atlas. For this article, and since we are most familiar with the MongoDB shell, let's select the "Connect with MongoDB Shell" option:



MongoDB will provide us with setup instructions for installing the MongoDB Shell and connecting to our cluster on our computer. Select the operating system you are using (e.g., macOS, Windows) and follow the directions to connect via a command line.

Connecting to the cluster should look similar to this:



> Note: Replace the <username> in the string that MongoDB provides with the username we created earlier.

We are now connected to our MongoDB Atlas cluster and can navigate our collections using the mongosh syntax we learned throughout this course!

Wrap Up

In this article, we learned about MongoDB Atlas and the tools it offers us to build production-ready databases with just a few clicks. Let's take a moment to go over some key takeaways:

- MongoDB Atlas is MongoDB's cloud toolset offering that allows us to store our data in MongoDB databases that run in the cloud.
- MongoDB Atlas provides various other solutions on top of a cloud database, including tools to perform analytics, visualization, and efficient searching.
- MongoDB Atlas manages our data within clusters.
- MongoDB Atlas allows us to connect to our cloud database on our local machine via a command line.

Using MongoDB Atlas will open new doors to how we can work with our databases. To learn more about MongoDB Atlas, check out these helpful resources:

- MongoDB: Clusters
- MongoDB University: A300 Atlas Security Course

Learn MongoDB: Next Steps

You've completed the Learn MongoDB course! What's next?

Congratulations, you've successfully completed the Learn MongoDB course! You gained fundamental experience in creating, managing, and retrieving data inside of a MongoDB database. More specifically, you learned:

- What MongoDB is and how you can use it to store data in a flexible format.
- How to execute commands in the `mongo` shell to interact with a MongoDB database instance.
- How to create new documents in a MongoDB collection, and query a collection for specific data.
- How to update and delete existing documents.
- How to create and use indexes to improve the efficiency of your queries.
- What data modeling is and how to create relationships between MongoDB data.
- What aggregation is and how to build an aggregation pipeline.
- How to deploy a MongoDB Atlas cluster.

Your learning journey into MongoDB isn't over yet! There are plenty of other topics that you can dive into to continue learning. Take a look at our recommendations for the next steps.

If you'd like to learn more about working with MongoDB, we recommend the following content:

- Check out the MongoDB Aggregation Framework course to get hands-on experience using MongoDB's powerful aggregation framework to transform and analyze data.
- Consider the MongoDB Data Modeling course if you want to learn techniques and patterns for building schemas for projects of all sizes.
- Browse the catalog of MongoDB University courses to see other useful content that you might be interested in as you continue your MongoDB journey!

If you are interested in learning about relational databases, we recommend the following courses:

- Check out the Learn SQL course to learn the fundamentals of the Structured Query Language, used to interact with and manage relational databases.
- Check out the Design Databases with PostgreSQL skill path if you are interested in learning the foundational principles of relational database design and architecture while developing a familiarity with the SQL programming language.

Once again, congratulations on finishing your Learn MongoDB course! We are excited to see what you accomplish next. Happy coding!