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:

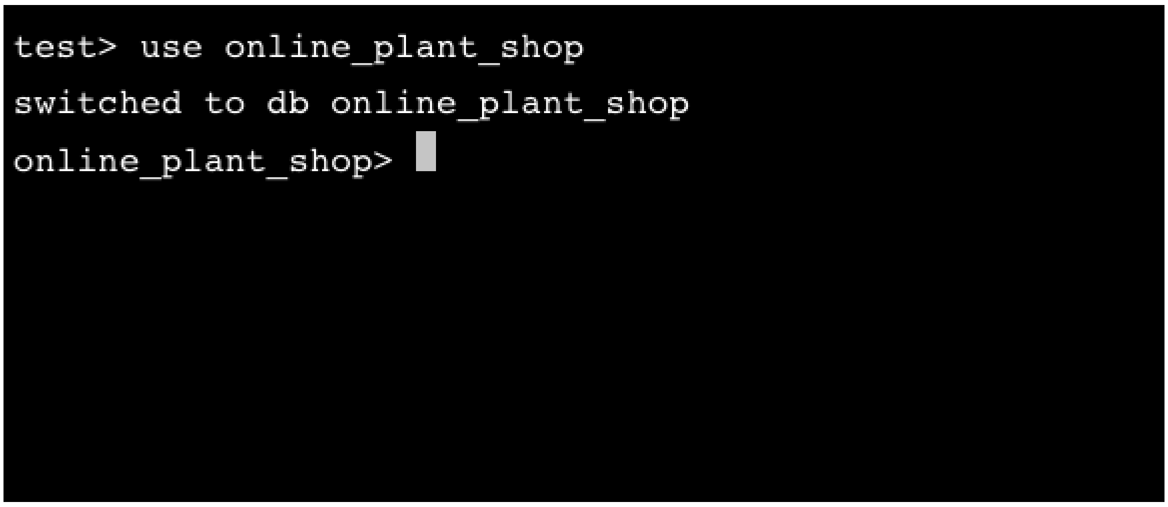
```
online_plant_shop        73.7 KiB
plant_lovers_meet        55.7 MiB
my_portfolio_site        9.57 MiB
admin                     340 KiB
local                    1.37 GiB
config                  12.00 KiB
```

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.

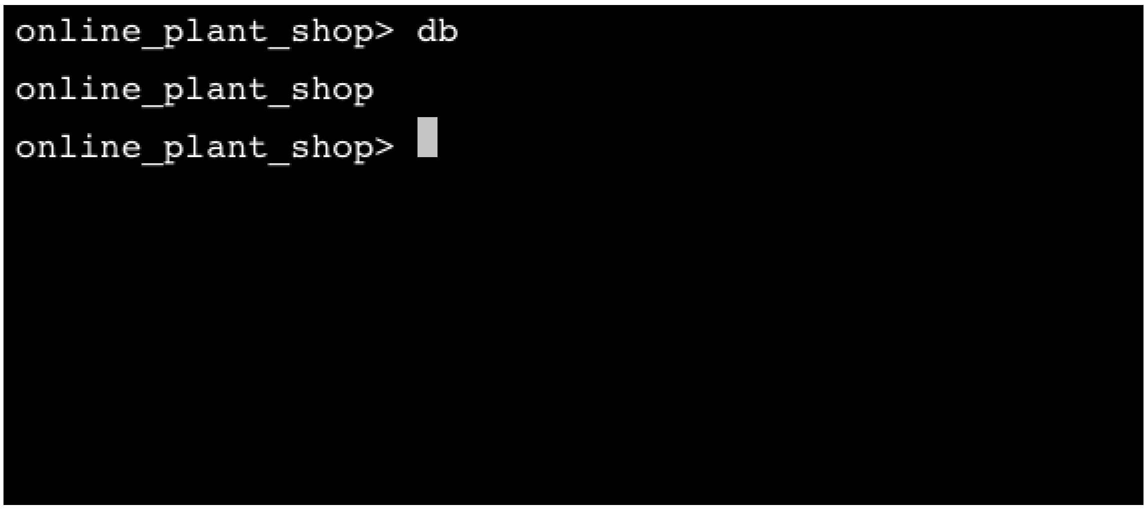
Here is what our terminal might look like:



```
test> use online_plant_shop
switched to db online_plant_shop
online_plant_shop> 
```

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:



```
online_plant_shop> db
online_plant_shop
online_plant_shop> 
```

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

Instructions

1.
Let's get familiar with our environment and orient ourselves by seeing what databases currently exist in our database instance.

Use the appropriate MongoDB command to see a list of all the current databases.

To check your commands for each task, use the **Check Work** button.

Note: Every exercise in this course will have the MongoDB Shell connected to a `test` database when it first loads.

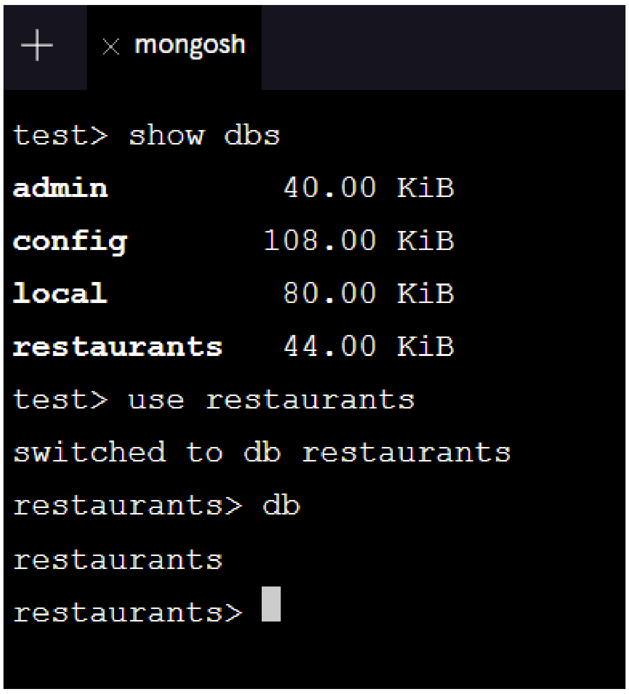
Hint
You can use the `show dbs` command to see a list of all your databases.

2.
Throughout this course, we will be working closely with the `restaurants` database. Navigate to the `restaurants` database in your MongoDB Shell.

Hint
You can use the `use <db>` command to navigate to a specific database.

3.
Before moving on, use the appropriate MongoDB command to confirm that you are in the `restaurants` database.

Hint
You can use the `db` command to see which database you are actively using. The output should return `restaurants`.



```
test> show dbs
admin          40.00 KiB
config        108.00 KiB
local          80.00 KiB
restaurants    44.00 KiB
test> use restaurants
switched to db restaurants
restaurants> db
restaurants
restaurants>
```

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!

Instructions

1.

Inside the `restaurants` database, there is a collection called `listingsAndReviews`.

Connect to the `restaurants` database, and then query the `listingsAndReviews` collection to get familiar with the documents it stores.

After running the command, be sure to hit the **Check Work** button!

Hint

Remember, you can query a collection using the following syntax:

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

To see a list of all collections in a database, you can type the command `show collections`.

Need another hint?

Your query should look as follows:

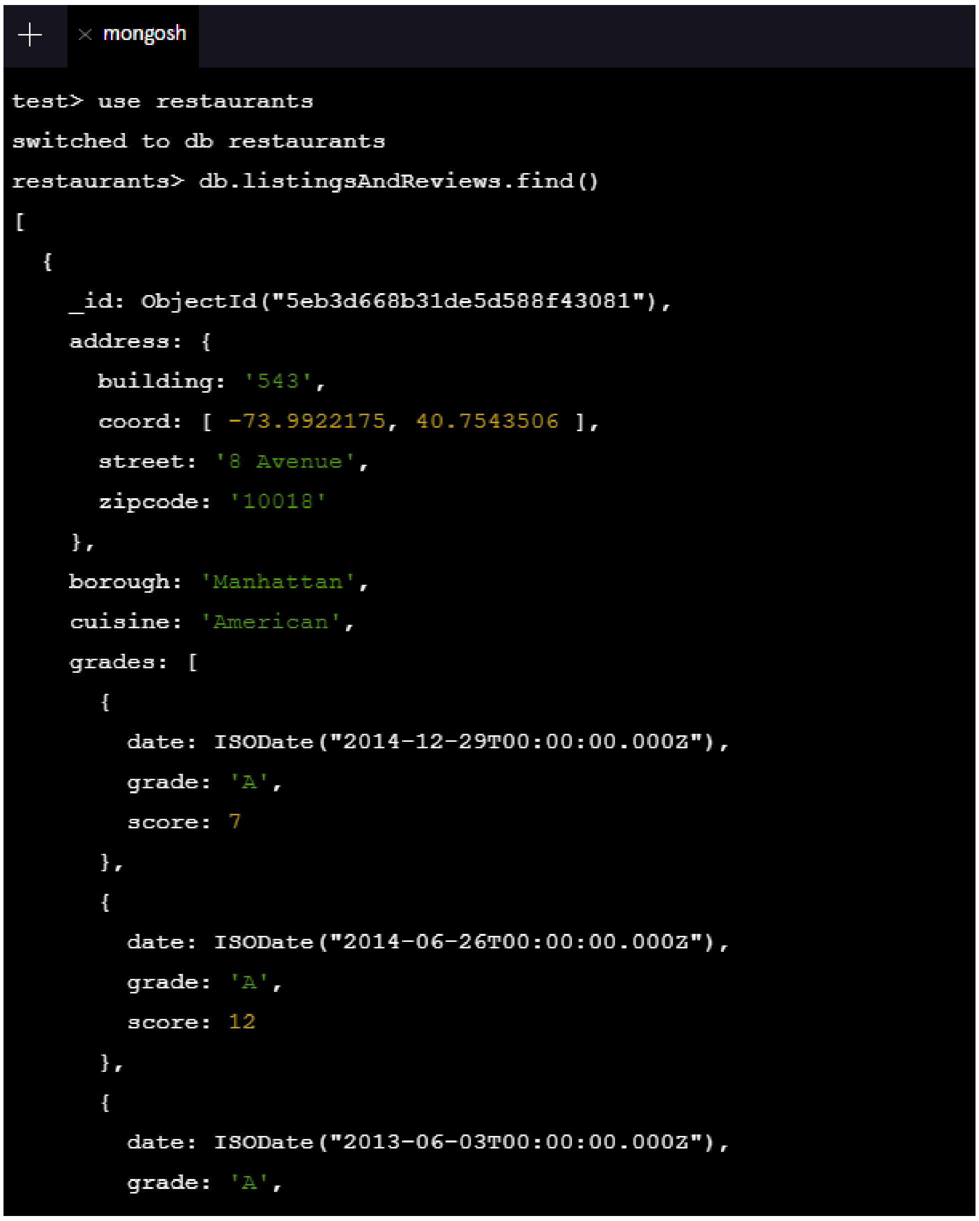
```
db.listingsAndReviews.find()
```

2.

The cursor only returned the first batch of documents. Iterate through the cursor to see the next batch of documents.

Hint

You can use the `it` command to iterate through your cursor.



```
test> use restaurants
switched to db restaurants
restaurants> db.listingsAndReviews.find()
[
  {
    _id: ObjectId("5eb3d668b31de5d588f43081"),
    address: {
      building: '543',
      coord: [ -73.9922175, 40.7543506 ],
      street: '8 Avenue',
      zipcode: '10018'
    },
    borough: 'Manhattan',
    cuisine: 'American',
    grades: [
      {
        date: ISODate("2014-12-29T00:00:00.000Z"),
        grade: 'A',
        score: 7
      },
      {
        date: ISODate("2014-06-26T00:00:00.000Z"),
        grade: 'A',
        score: 12
      },
      {
        date: ISODate("2013-06-03T00:00:00.000Z"),
        grade: 'A',
```

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 `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 `"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!

Instructions

1.

You're visiting Brooklyn, New York, and want to find a place for lunch. Thankfully, we have a database full of restaurants!

Connect to the `restaurants` collection, and then query the `listingsAndReviews` collection to find a list of restaurants in the `borough` `"Brooklyn"`.

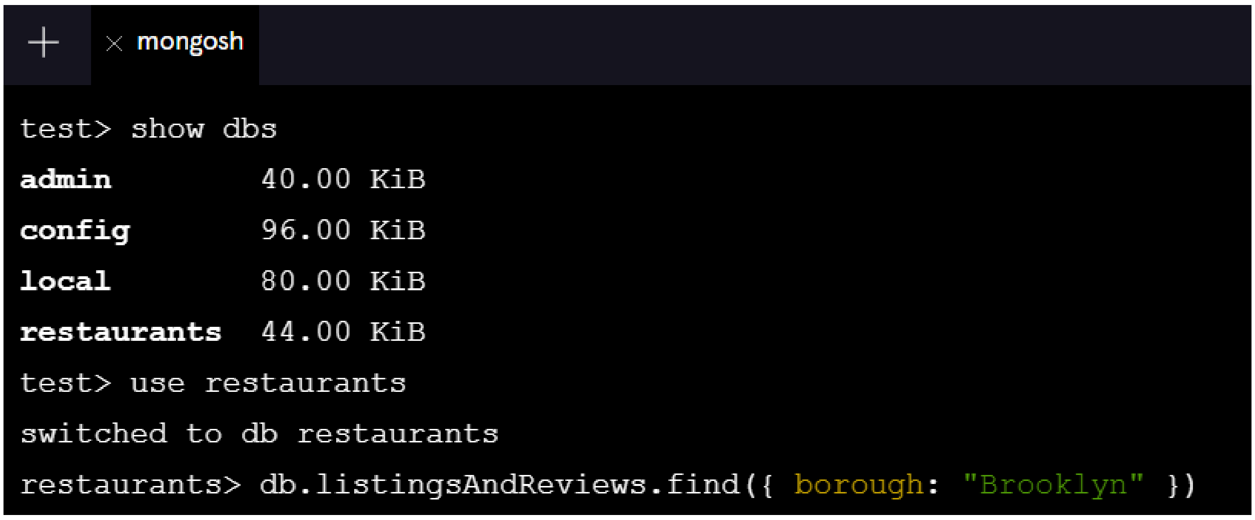
Press the **Check Work** button to move on to the next task.

Hint
You can use the `.find()` method with the query `{borough: "Brooklyn"}` to find a list of Brooklyn restaurants.

Wow! There are so many options! Narrow down your results by querying the `listingsAndReviews` collection again, this time where the `borough` is `"Brooklyn"` and `cuisine` is `"Caribbean"`.

Press the **Check Work** button again to complete the instructions for this exercise.

Hint
You can use the `.find()` method with the filter `{borough: "Brooklyn", cuisine: "Caribbean"}` to find a list of Caribbean restaurants in Brooklyn.



```
test> show dbs
admin          40.00 KiB
config         96.00 KiB
local          80.00 KiB
restaurants    44.00 KiB
test> use restaurants
switched to db restaurants
restaurants> db.listingsAndReviews.find({ borough: "Brooklyn" })
```

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!

Instructions

1.

Let’s return to our `restaurants` database. Switch to the database and query it to see *all* of the records inside the `listingsAndReviews` collection. See if you can spot all of the embedded documents!

Hint

Remember, you can query a collection using the following syntax:

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

To see a list of all collections in a database, you can type the command `show collections`.

2.

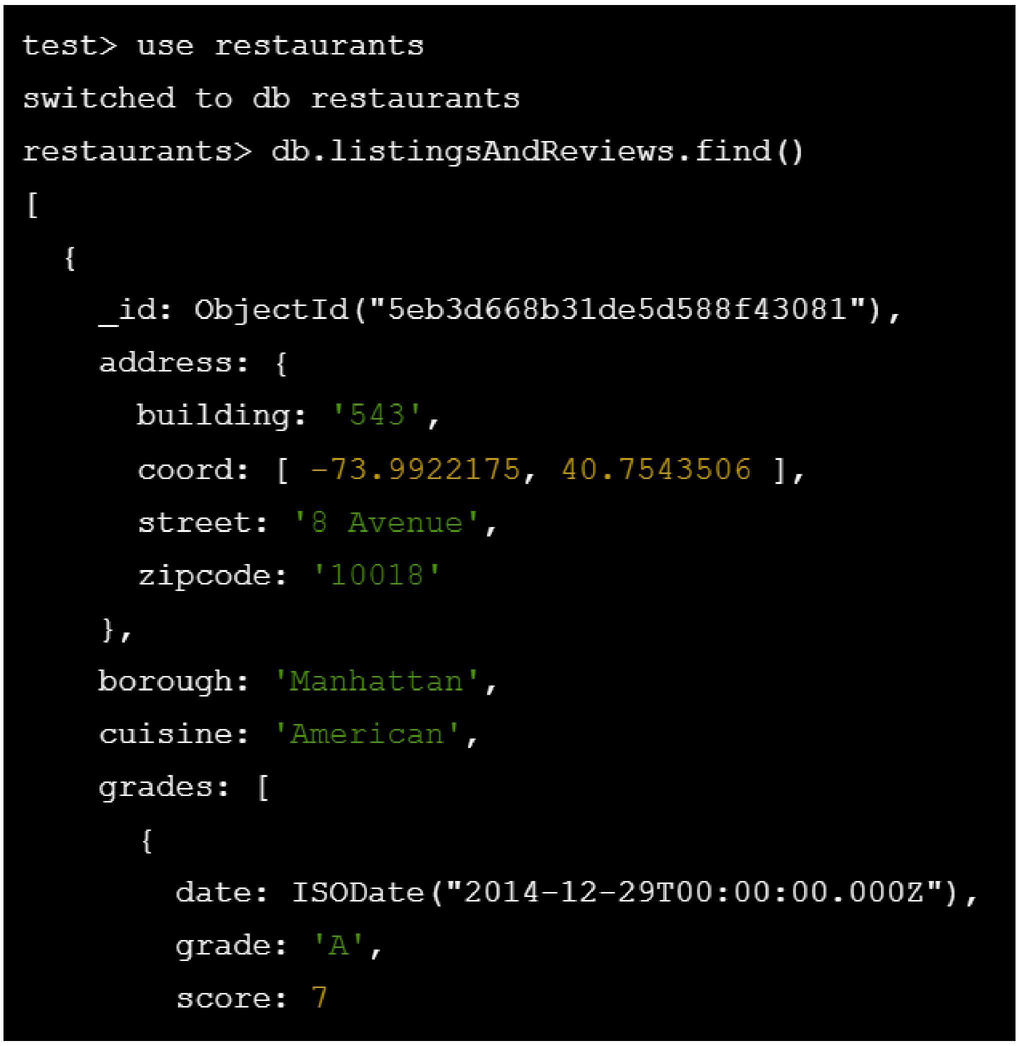
Notice that the restaurant listings in our collection have an embedded document called `address`. We are in a rush and looking for a meal close to where we are visiting in Brooklyn, New York.

Query the `listingsAndReviews` collection for restaurants where the `zipcode` is `11231`.

Hint

The `zipcode` field is in the `address` embedded document. Recall that to query embedded documents, we must use dot notation (`.`) and wrap the fields in quotation marks.

mongosh



```
test> use restaurants
switched to db restaurants
restaurants> db.listingsAndReviews.find()
[
  {
    _id: ObjectId("5eb3d668b31de5d588f43081"),
    address: {
      building: '543',
      coord: [ -73.9922175, 40.7543506 ],
      street: '8 Avenue',
      zipcode: '10018'
    },
    borough: 'Manhattan',
    cuisine: 'American',
    grades: [
      {
        date: ISODate("2014-12-29T00:00:00.000Z"),
        grade: 'A',
        score: 7
```

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 `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!

Instructions

1.

Connect to the `restaurants` database, then query the `listingsAndReviews` collection to retrieve a list of restaurants where the `restaurant_id` is greater than `"50000000"`.

Hint

You can use the `.find()` method with the `$gt` operator to find all the restaurants whose `restaurant_id` is greater than `"50000000"`. Remember to use the following syntax:

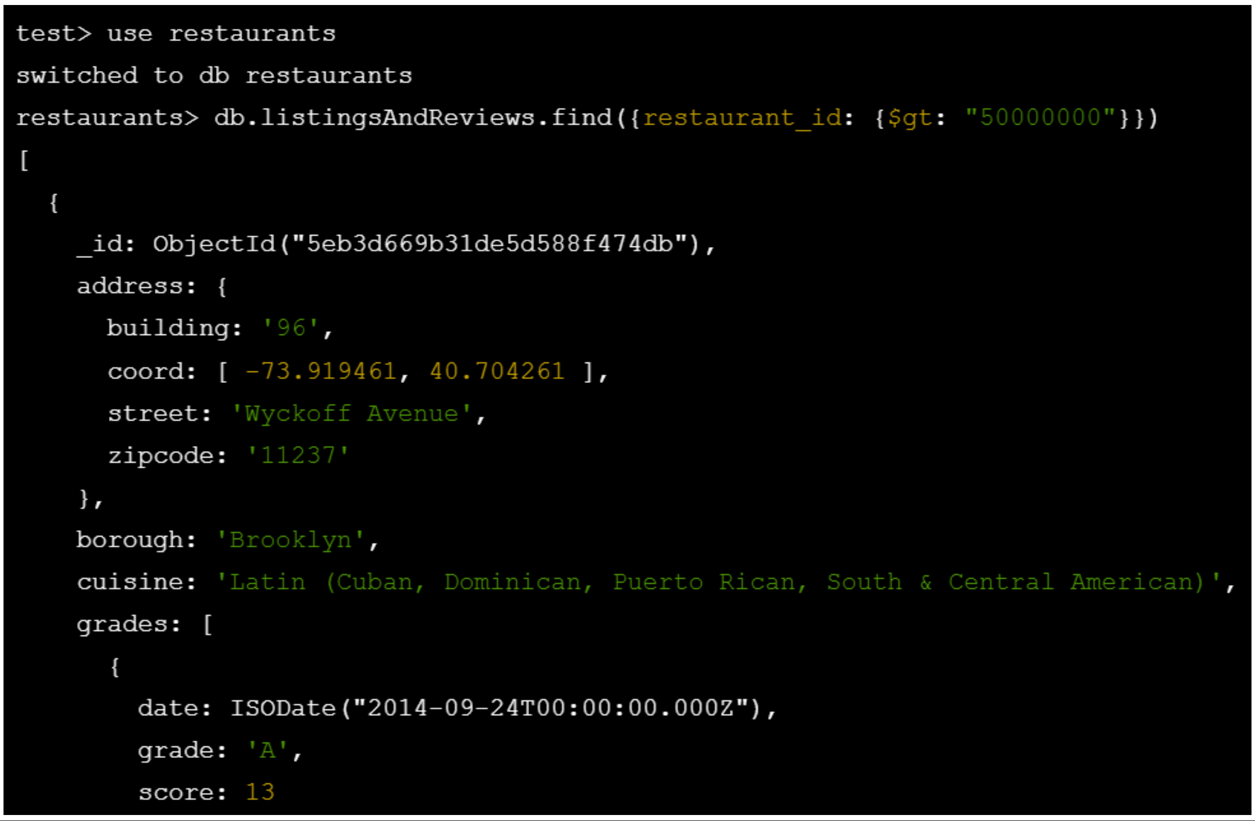
```
db.<collection>.find({ <field>: { $gt: <value> } })
```

Hint

The `street` field is an embedded document inside of the `address` field. You can query the embedded field using the `.find()` method with the `$lte` operator like so:

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

mongosh



```
test> use restaurants
switched to db restaurants
restaurants> db.listingsAndReviews.find({restaurant_id: {$gt: "50000000"}})
[
  {
    _id: ObjectId("5eb3d669b31de5d588f474db"),
    address: {
      building: '96',
      coord: [ -73.919461, 40.704261 ],
      street: 'Wyckoff Avenue',
      zipcode: '11237'
    },
    borough: 'Brooklyn',
    cuisine: 'Latin (Cuban, Dominican, Puerto Rican, South & Central American)',
    grades: [
      {
        date: ISODate("2014-09-24T00:00:00.000Z"),
        grade: 'A',
        score: 13
```

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 `.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 `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 `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 `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.

Before moving on, let's practice using the `.sort()` method to sort our queries.

Instructions

1.

Connect to the `restaurants` database, then query the `listingsAndReviews` collection, to retrieve a list of restaurants where the `cuisine` is `"Spanish"`. The query results should be sorted by the `name` field alphabetically.

Hint

You can use the `.find()` method with the filter, `{cuisine: "Spanish}`, to query the collection. Append the `.sort()` method with the correct field and value to your query to return sorted results. Remember that a value of `1` sorts chronologically or alphabetically, while `-1` sorts in the reverse order.

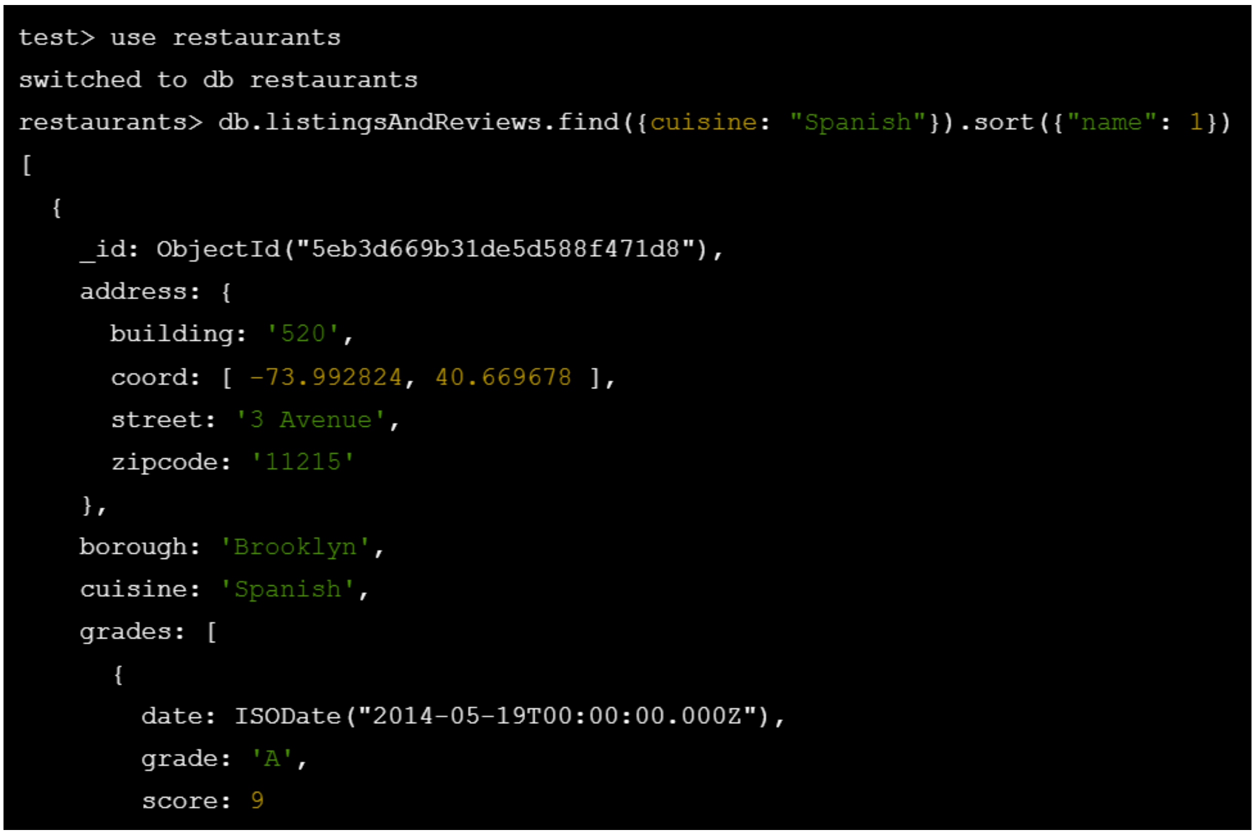
2.

Query the same collection, `listingsAndReviews`, to return a list of restaurants where the `borough` is `"Queens"`. The results should be sorted by the address `zipcode` field, in descending order.

Hint

To sort an embedded document in descending order using the following syntax:

mongosh



```
test> use restaurants
switched to db restaurants
restaurants> db.listingsAndReviews.find({cuisine: "Spanish"}).sort({"name": 1})
[
  {
    _id: ObjectId("5eb3d669b31de5d588f471d8"),
    address: {
      building: '520',
      coord: [ -73.992824, 40.669678 ],
      street: '3 Avenue',
      zipcode: '11215'
    },
    borough: 'Brooklyn',
    cuisine: 'Spanish',
    grades: [
      {
        date: ISODate("2014-05-19T00:00:00.000Z"),
        grade: 'A',
        score: 9
```

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 **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 `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 `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 `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 `_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 })
```

Before we wrap up the lesson, let's practice writing queries with projections!

Instructions

1.

Connect to the `restaurants` database, then using the same `listingsAndReviews` collection as the previous exercises, run a query to get a list of restaurants where the `borough` is `"Bronx"`. Use a projection to return the `_id`, `name`, and `cuisine` fields from each matching document.

Hint

Remember that to include a projection, you can pass in a second argument to `.find()` that is a document containing the fields you want to include or exclude. Provide a value of `1` to include that field, and a value of `0` to exclude it.

Query the `listingsAndReviews` collection again. This time, your query should get all documents in the collection and return all fields except for the `address` and `grades` fields.

Hint

Remember that you can query all documents in a collection by passing using the `.find()` method and passing an empty document as its first argument.

mongosh

```
test> use restaurants
switched to db restaurants
restaurants> show collections
listingsAndReviews
restaurants> db.listingsAndReviews.find({borough: "Bronx"}, {name: 1, cuisine: 1})
[
  {
    _id: ObjectId("5eb3d669b31de5d588f477aa"),
    cuisine: 'Juice, Smoothies, Fruit Salads',
    name: "D' Licua Fruit & Juice"
  },
  {
    _id: ObjectId("5eb3d669b31de5d588f47a2f"),
    cuisine: 'Chinese',
    name: 'Golden China Pavilion'
  },
```

Review

Nice job! In this lesson, we learned how to query documents in MongoDB. Let's recap some key takeaways from the lesson:

- 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.

In addition to the methods and operators we've covered in this lesson, MongoDB provides us with even more syntax that can be useful to us when performing queries:

- The `.count()` method returns the number of documents that match a query.
- The `.limit()` method can be chained to the `.find()` method, and specifies the maximum number of documents a query will output.
- The `$exists` operator can be included in a query filter to only match documents that contain the given field.
- The `$ne` operator helps check if a field is not equal to a specified value.
- The `$and` and `$or` operators help perform AND or OR logic operators.

Lastly, if you are looking for a way to make query outputs look a bit more "pretty", you can use the `.pretty()` method!

Instructions

We have provided you with the `listingsAndReviews` collection. Before moving on, spend some time experimenting with writing queries, using the syntax you learned throughout this lesson. If you are up for a challenge, try any of the following tasks listed below. Remember to first connect to the `restaurants` database to access the `listingsAndReviews` collection. Good luck, and click **Up Next** when you are ready to move on!

Optional Tasks:

- Find all of the restaurants in the borough `"Queens"` that serves `"Japanese"` cuisine sorted in reverse alphabetical order by name.
- Count the number of restaurants in the borough `"Manhattan"` serving your favorite cuisine, limited to five results.
- Find all the restaurants in the borough `"Bronx"` that serve one of the following cuisines: `"Juice, Smoothies, Fruit Salads"`, `"Spanish"`, or `"Pizza"`.