MongoDB Overview
- Database Type: NoSQL document database.
- Data Structure: Uses collections and documents (BSON format).

Query API
- Basic Operations:
  - Create DB: Use `use dbName` to create or switch to a database.
  - Create Collection: `db.createCollection("collectionName")`.
  - Insert: `db.collectionName.insertOne({...})` or `insertMany([...])`.
  - Find: `db.collectionName.find({...})`, with options for projections and sorting.

- Update: `db.collectionName.updateOne(filter, update)` or `updateMany(filter, update)`.
- Delete: `db.collectionName.deleteOne(filter)` or `deleteMany(filter)`.

Query Operators

- Comparison Operators: `$eq`, `$gt`, `$gte`, `$lt`, `$lte`, `$ne`, etc.
- Logical Operators: `$and`, `$or`, `$not`, `$nor`.
- Element Operators: `$exists`, `$type`.
- Array Operators: `$all`, `$elemMatch`, `$size`.

Indexing

- Purpose: Improve query performance.
- Types: Single field, compound, text, geospatial indexes.
- Create Index: `db.collectionName.createIndex({ field: 1 })`.

Validation
- Schema Validation: Define rules for documents in a collection using JSON schema.
- Example: `db.createCollection("collectionName", { validator: { $jsonSchema: { / schema / } } })`.

Data API

- RESTful API: Access MongoDB data over HTTP.
- Use Cases: Integrate with web applications, mobile apps, etc.

Drivers
- Purpose: Allow different programming languages to interact with MongoDB.
- Node.js Driver: Official driver for Node.js.
  - Installation: `npm install mongodb`.
  - Basic Usage:
    ```javascript
    const { MongoClient } = require('mongodb');
    const client = new MongoClient('mongodb://localhost:27017');
    ```

```
   await client.connect();
   const db = client.db('dbName');
   const collection =
db.collection('collectionName');
```

 Charts
- MongoDB Charts: Visualization tool for MongoDB data.
- Usage: Create dashboards and visual reports.

This should give you a solid framework to explore MongoDB further. If you need more details on any specific topic, just let me know!

How MongoDB Stores Data

Before we go any further, let's take a moment to understand how data is stored in MongoDB.
MongoDB stores data in BSON documents. BSON is a binary representation of JSON (JavaScript Object Notation) documents. When

you read MongoDB documentation, you'll frequently see the term "document," but you can think of a document as simply a JavaScript object. For those coming from the SQL world, you can think of a document as being roughly equivalent to a row.
MongoDB stores groups of documents in collections. For those with a SQL background, you can think of a collection as being roughly equivalent to a table.
Every document is required to have a field named `_id`. The value of `_id` must be unique for each document in a collection, is immutable, and can be of any type other than an array. MongoDB will automatically create an index on `_id`. You can choose to make the value of `_id` meaningful (rather than a somewhat random ObjectId) if you have a unique value for each document that you'd like to be able to quickly search.
In this blog series, we'll use the sample Airbnb listings dataset. The `sample_airbnb` database contains one collection: `listingsAndReviews`. This collection contains documents about Airbnb listings and their reviews.
Let's take a look at a document in the `listingsAndReviews` collection. Below is part of an Extended JSON representation of a BSON document:

```
{
"_id": "10057447",
"listing_url": "https://www.airbnb.com/rooms/10057447",
"name": "Modern Spacious 1 Bedroom Loft",
"summary": "Prime location, amazing lighting and no annoying neighbours. Good place to rent if you want a relaxing time in Montreal.",
"property_type": "Apartment",
"bedrooms": {"$numberInt":"1"},
"bathrooms": {"$numberDecimal":"1.0"},
"amenities": ["Internet","Wifi","Kitchen","Heating","Family/kid friendly","Washer","Dryer","Smoke detector","First aid kit","Safety card","Fire extinguisher","Essentials","Shampoo","24-hour check-in","Hangers","Iron","Laptop friendly workspace"],
}
```

For more information on how MongoDB stores data, see the MongoDB Back to Basics Webinar that I co-hosted with Ken Alger.

Setup

To make following along with this blog post easier, I've created a starter template for a Node.js script that accesses an Atlas cluster.

1. Download a copy of template.js.
2. Open `template.js` in your favorite code editor.
3. Update the Connection URI to point to your Atlas cluster. If you're not sure how to do that, refer back to the first post in this series.
4. Save the file as `crud.js`.

You can run this file by executing `node crud.js` in your shell. At this point, the file simply opens and closes a connection to your Atlas cluster, so no output is expected. If you see DeprecationWarnings, you can ignore them for the purposes of this post.

Create

Now that we know how to connect to a MongoDB database and we understand how data is stored in a MongoDB database, let's create some data!

Create One Document

Let's begin by creating a new Airbnb listing. We can do so by calling Collection's insertOne(). `insertOne()` will insert a single document into the collection. The only required parameter is the new document (of type object) that will be inserted. If our new document does not contain the `_id` field, the MongoDB driver will automatically create an `_id` for the document.

Our function to create a new listing will look something like the following:

```
async function createListing(client, newListing){
const result = await client.db("sample_airbnb").collection("listingsAndReviews").insertOne(newListing);
console.log(`New listing created with the following id: ${result.insertedId}`);
}
```

We can call this function by passing a connected MongoClient as well as an object that contains information about a listing.

```
await createListing(client,
{
name: "Lovely Loft",
summary: "A charming loft in Paris",
bedrooms: 1,
bathrooms: 1
}
);
```

The output would be something like the following:

```
New listing created with the following id: 5d9ddadee415264e135ccec8
```

Note that since we did not include a field named `_id` in the document, the MongoDB driver automatically created an `_id` for us. The `_id` of the document you create will be different from the one shown above. For more information on how MongoDB generates `_id`, see Quick Start: BSON Data Types - ObjectId.

If you're not a fan of copying and pasting, you can get a full copy of the code above in the Node.js Quick Start GitHub Repo.

Create Multiple Documents

Sometimes you will want to insert more than one document at a time. You could choose to repeatedly call `insertOne()`. The problem is that, depending on how you've structured your code, you may end up waiting for each insert operation to return before beginning the next, resulting in slow code.

Instead, you can choose to call Collection's insertMany(). `insertMany()` will insert an array of documents into your collection.

One important option to note for `insertMany()` is `ordered`. If `ordered` is set to `true`, the documents will be inserted in the order given in the array. If any of the inserts fail (for example, if you attempt to insert a document with an `_id` that is already being used by another document in the collection), the remaining documents will not be inserted. If ordered is set to `false`, the documents may not be inserted in the order given in the array. MongoDB will attempt to insert all of the documents in the given array—regardless of whether any of the other inserts fail. By default, `ordered` is set to `true`.

Let's write a function to create multiple Airbnb listings.

```
async function createMultipleListings(client, newListings){
```

```
const result = await client.db("sample_airbnb").collection("listingsAndReviews").insertMany(newListings);

console.log(`${result.insertedCount} new listing(s) created with the following id(s):`);
console.log(result.insertedIds);
}
```

We can call this function by passing a connected MongoClient and an array of objects that contain information about listings.

```
await createMultipleListings(client, [
{
name: "Infinite Views",
summary: "Modern home with infinite views from the infinity pool",
property_type: "House",
bedrooms: 5,
bathrooms: 4.5,
beds: 5
},
{
name: "Private room in London",
property_type: "Apartment",
bedrooms: 1,
bathroom: 1
},
{
name: "Beautiful Beach House",
summary: "Enjoy relaxed beach living in this house with a private beach",
bedrooms: 4,
bathrooms: 2.5,
beds: 7,
last_review: new Date()
}
]);
```

Note that every document does not have the same fields, which is perfectly OK. (I'm guessing that those who come from the SQL world will find this incredibly uncomfortable, but it really will be OK 😊.) When you use MongoDB, you get a lot of flexibility in how to structure your documents. If you later decide you want to add schema validation rules so you can guarantee your documents have a particular structure, you can.

The output of calling `createMultipleListings()` would be something like the following:

```
3 new listing(s) created with the following id(s):
{
'0': 5d9ddadee415264e135ccec9,
'1': 5d9ddadee415264e135cceca,
'2': 5d9ddadee415264e135ccecb
}
```

Just like the MongoDB Driver automatically created the `_id` field for us when we called `insertOne()`, the Driver has once again created the `_id` field for us when we called `insertMany()`.

If you're not a fan of copying and pasting, you can get a full copy of the code above in the Node.js Quick Start GitHub Repo.

Read

Now that we know how to **create** documents, let's **read** one!

Read One Document

Let's begin by querying for an Airbnb listing in the listingsAndReviews collection by name.

We can query for a document by calling Collection's findOne(). `findOne()` will return the first document that matches the given query. Even if more than one document matches the query, only one document will be returned.

`findOne()` has only one required parameter: a query of type object. The query object can contain zero or more properties that MongoDB will use to find a document in the collection. If you want to query all documents in a collection without narrowing your results in any way, you can simply send an empty object.

Since we want to search for an Airbnb listing with a particular name, we will include the name field in the query object we pass to `findOne()`:

```
findOne({ name: nameOfListing })
```

Our function to find a listing by querying the name field could look something like the following:

```
async function findOneListingByName(client, nameOfListing) {
const result = await
client.db("sample_airbnb").collection("listingsAndReviews").findOne({ name:
nameOfListing });

if (result) {
```

```
console.log(`Found a listing in the collection with the name
'${nameOfListing}':`);
console.log(result);
} else {
console.log(`No listings found with the name '${nameOfListing}'`);
}
}
```

We can call this function by passing a connected MongoClient as well as the name of a listing we want to find. Let's search for a listing named "Infinite Views" that we created in an earlier section.

```
await findOneListingByName(client, "Infinite Views");
```

The output should be something like the following.

```
Found a listing in the collection with the name 'Infinite Views':
{
_id: 5da9b5983e104518671ae128,
name: 'Infinite Views',
summary: 'Modern home with infinite views from the infinity pool',
property_type: 'House',
bedrooms: 5,
bathrooms: 4.5,
beds: 5
}
```

Note that the `_id` of the document in your database will not match the `_id` in the sample output above.

If you're not a fan of copying and pasting, you can get a full copy of the code above in the Node.js Quick Start GitHub Repo.

Read Multiple Documents

Now that you know how to query for one document, let's discuss how to query for multiple documents at a time. We can do so by calling Collection's find().

Similar to `findOne()`, the first parameter for `find()` is the query object. You can include zero to many properties in the query object.

Let's say we want to search for all Airbnb listings that have minimum numbers of bedrooms and bathrooms. We could do so by making a call like the following:

```
client.db("sample_airbnb").collection("listingsAndReviews").find(
{
bedrooms: { $gte: minimumNumberOfBedrooms },
bathrooms: { $gte: minimumNumberOfBathrooms }
}
);
```

As you can see above, we have two properties in our query object: one for bedrooms and one for bathrooms. We can leverage the $gte comparison query operator to search for documents that have bedrooms greater than or equal to a given number. We can do the same to satisfy our minimum number of bathrooms requirement. MongoDB provides a variety of other comparison query operators that you can utilize in your queries. See the official documentation for more details.

The query above will return a Cursor. A Cursor allows traversal over the result set of a query.

You can also use Cursor's functions to modify what documents are included in the results. For example, let's say we want to sort our results so that those with the most recent reviews are returned first. We could use Cursor's sort() function to sort the results using the `last_review` field. We could sort the results in descending order (indicated by passing -1 to `sort()`) so that listings with the most recent reviews will be returned first. We can now update our existing query to look like the following.

```
const cursor =
client.db("sample_airbnb").collection("listingsAndReviews").find(
{
bedrooms: { $gte: minimumNumberOfBedrooms },
bathrooms: { $gte: minimumNumberOfBathrooms }
}
).sort({ last_review: -1 });
```

The above query matches 192 documents in our collection. Let's say we don't want to process that many results inside of our script. Instead, we want to limit our results to a smaller number of documents. We can chain another of `sort()`'s functions to our existing query: limit(). As the name implies, `limit()` will set the limit for the cursor. We can now update our query to only return a certain number of results.

```
const cursor =
client.db("sample_airbnb").collection("listingsAndReviews").find(
{
bedrooms: { $gte: minimumNumberOfBedrooms },
bathrooms: { $gte: minimumNumberOfBathrooms }
}
).sort({ last_review: -1 })
.limit(maximumNumberOfResults);
```

We could choose to iterate over the cursor to get the results one by one. Instead, if we want to retrieve all of our results in an array, we can call Cursor's toArray() function. Now our code looks like the following:

```
const cursor = client.db("sample_airbnb").collection("listingsAndReviews").find(
{
bedrooms: { $gte: minimumNumberOfBedrooms },
bathrooms: { $gte: minimumNumberOfBathrooms }
}
).sort({ last_review: -1 })
.limit(maximumNumberOfResults);
const results = await cursor.toArray();
```

Now that we have our query ready to go, let's put it inside an asynchronous function and add functionality to print the results.

```
async function findListingsWithMinimumBedroomsBathroomsAndMostRecentReviews(client, {
minimumNumberOfBedrooms = 0,
minimumNumberOfBathrooms = 0,
maximumNumberOfResults = Number.MAX_SAFE_INTEGER
} = {}) {
const cursor = client.db("sample_airbnb").collection("listingsAndReviews").find(
{
bedrooms: { $gte: minimumNumberOfBedrooms },
bathrooms: { $gte: minimumNumberOfBathrooms }
}
).sort({ last_review: -1 })
.limit(maximumNumberOfResults);

const results = await cursor.toArray();

if (results.length > 0) {
console.log(`Found listing(s) with at least ${minimumNumberOfBedrooms} bedrooms and ${minimumNumberOfBathrooms} bathrooms:`);
results.forEach((result, i) => {
date = new Date(result.last_review).toDateString();

console.log();
console.log(`${i + 1}. name: ${result.name}`);
console.log(` _id: ${result._id}`);
console.log(` bedrooms: ${result.bedrooms}`);
console.log(` bathrooms: ${result.bathrooms}`);
console.log(` most recent review date: ${new Date(result.last_review).toDateString()}`);
});
} else {
console.log(`No listings found with at least ${minimumNumberOfBedrooms}
bedrooms and ${minimumNumberOfBathrooms} bathrooms`);
}
}
```

We can call this function by passing a connected MongoClient as well as an object with properties indicating the minimum number of bedrooms, the minimum number of bathrooms, and the maximum number of results.

```
await
findListingsWithMinimumBedroomsBathroomsAndMostRecentReviews(client, {
minimumNumberOfBedrooms: 4,
minimumNumberOfBathrooms: 2,
maximumNumberOfResults: 5
});
```

If you've created the documents as described in the earlier section, the output would be something like the following:

```
Found listing(s) with at least 4 bedrooms and 2 bathrooms:

1. name: Beautiful Beach House
_id: 5db6ed14f2e0a60683d8fe44
bedrooms: 4
bathrooms: 2.5
most recent review date: Mon Oct 28 2019

2. name: Spectacular Modern Uptown Duplex
_id: 582364
bedrooms: 4
bathrooms: 2.5
most recent review date: Wed Mar 06 2019

3. name: Grace 1 - Habitat Apartments
_id: 29407312
bedrooms: 4
bathrooms: 2.0
most recent review date: Tue Mar 05 2019

4. name: 6 bd country living near beach
_id: 2741869
bedrooms: 6
bathrooms: 3.0
most recent review date: Mon Mar 04 2019

```

```
5. name: Awesome 2-storey home Bronte Beach next to Bondi!
_id: 20206764
bedrooms: 4
bathrooms: 2.0
most recent review date: Sun Mar 03 2019
```

If you're not a fan of copying and pasting, you can get a full copy of the code above in the Node.js Quick Start GitHub Repo.

Update

We're halfway through the CRUD operations. Now that we know how to **create** and **read** documents, let's discover how to **update** them.

Update One Document

Let's begin by updating a single Airbnb listing in the listingsAndReviews collection.

We can update a single document by calling Collection's updateOne(). `updateOne()` has two required parameters:

1. `filter` (object): the Filter used to select the document to update. You can think of the filter as essentially the same as the query param we used in findOne() to search for a particular document. You can include zero properties in the filter to search for all documents in the collection, or you can include one or more properties to narrow your search.
2. `update` (object): the update operations to be applied to the document. MongoDB has a variety of update operators you can use such as `$inc`, `$currentDate`, `$set`, and `$unset` among others. See the official documentation for a complete list of update operators and their descriptions.

`updateOne()` also has an optional `options` param. See the updateOne() docs for more information on these options.

`updateOne()` will update the first document that matches the given query. Even if more than one document matches the query, only one document will be updated.

Let's say we want to update an Airbnb listing with a particular name. We can use `updateOne()` to achieve this. We'll include the name of the listing in the filter param. We'll use the $set update operator to set new values for new or existing fields in the document we are updating. When we use `$set`, we pass a document that contains fields and values that should be updated or created. The document that we pass to `$set` will not replace the existing document; any fields that are part of the original document but not part of the document we pass to `$set` will remain as they are.

Our function to update a listing with a particular name would look like the following:

```
async function updateListingByName(client, nameOfListing, updatedListing) {
const result = await
client.db("sample_airbnb").collection("listingsAndReviews")
.updateOne({ name: nameOfListing }, { $set: updatedListing });

console.log(`${result.matchedCount} document(s) matched the query
criteria.`);
console.log(`${result.modifiedCount} document(s) was/were updated.`);
}
```

Let's say we want to update our Airbnb listing that has the name "Infinite Views." We created this listing in an earlier section.

```
{
_id: 5db6ed14f2e0a60683d8fe42,
name: 'Infinite Views',
summary: 'Modern home with infinite views from the infinity pool',
property_type: 'House',
bedrooms: 5,
bathrooms: 4.5,
beds: 5
}
```

We can call `updateListingByName()` by passing a connected MongoClient, the name of the listing, and an object containing the fields we want to update and/or create.

```
await updateListingByName(client, "Infinite Views", { bedrooms: 6, beds: 8 });
```

Executing this command results in the following output.

```
1 document(s) matched the query criteria.
1 document(s) was/were updated.
```

Now our listing has an updated number of bedrooms and beds.

```
{
_id: 5db6ed14f2e0a60683d8fe42,
name: 'Infinite Views',
summary: 'Modern home with infinite views from the infinity pool',
property_type: 'House',
bedrooms: 6,
bathrooms: 4.5,
beds: 8
}
```

If you're not a fan of copying and pasting, you can get a full copy of the code above in the Node.js Quick Start GitHub Repo.

Upsert One Document

One of the options you can choose to pass to `updateOne()` is upsert. Upsert is a handy feature that allows you to update a document if it exists or insert a document if it does not.

For example, let's say you wanted to ensure that an Airbnb listing with a particular name had a certain number of bedrooms and bathrooms. Without upsert, you'd first use `findOne()` to check if the document existed. If the document existed, you'd use `updateOne()` to update the document. If the document did not exist, you'd use `insertOne()` to create the document. When you use upsert, you can combine all of that functionality into a single command.

Our function to upsert a listing with a particular name can be basically identical to the function we wrote above with one key difference: We'll pass `{upsert: true}` in the `options` param for `updateOne()`.

```
async function upsertListingByName(client, nameOfListing, updatedListing) {
const result = await
client.db("sample_airbnb").collection("listingsAndReviews")
.updateOne({ name: nameOfListing },
{ $set: updatedListing },
{ upsert: true });
console.log(`${result.matchedCount} document(s) matched the query
criteria.`);

if (result.upsertedCount > 0) {
console.log(`One document was inserted with the id ${result.upsertedId._id}`);
} else {
console.log(`${result.modifiedCount} document(s) was/were updated.`);
}
}
```

Let's say we aren't sure if a listing named "Cozy Cottage" is in our collection or, if it does exist, if it holds old data. Either way, we want to ensure the listing that exists in our collection has the most up-to-date data. We can call `upsertListingByName()` with a connected MongoClient, the name of the listing, and an object containing the up-to-date data that should be in the listing.

```
await upsertListingByName(client, "Cozy Cottage", { name: "Cozy Cottage", bedrooms: 2, bathrooms: 1 });
```

If the document did not previously exist, the output of the function would be something like the following:

```
0 document(s) matched the query criteria.
One document was inserted with the id 5db9d9286c503eb624d036a1
```

We have a new document in the listingsAndReviews collection:

```
{
_id: 5db9d9286c503eb624d036a1,
name: 'Cozy Cottage',
bathrooms: 1,
bedrooms: 2
}
```

If we discover more information about the "Cozy Cottage" listing, we can use `upsertListingByName()` again.

```
await upsertListingByName(client, "Cozy Cottage", { beds: 2 });
```

And we would see the following output.

```
1 document(s) matched the query criteria.
1 document(s) was/were updated.
```

Now our document has a new field named "beds."

```
{
_id: 5db9d9286c503eb624d036a1,
name: 'Cozy Cottage',
bathrooms: 1,
bedrooms: 2,
beds: 2
}
```

If you're not a fan of copying and pasting, you can get a full copy of the code above in the Node.js Quick Start GitHub Repo.

Update Multiple Documents

Sometimes you'll want to update more than one document at a time. In this case, you can use Collection's updateMany().

Like `updateOne()`, `updateMany()` requires that you pass a filter of type object and an update of type object. You can choose to include options of type object as well.

Let's say we want to ensure that every document has a field named `property_type`. We can use the $exists query operator to search for documents where the `property_type` field does not exist. Then we can use the $set update operator to set the `property_type` to "Unknown" for those documents. Our function will look like the following.

```
async function updateAllListingsToHavePropertyType(client) {
const result = await
client.db("sample_airbnb").collection("listingsAndReviews")
.updateMany({ property_type: { $exists: false } },
{ $set: { property_type: "Unknown" } });
console.log(`${result.matchedCount} document(s) matched the query
criteria.`);
console.log(`${result.modifiedCount} document(s) was/were updated.`);
}
```

We can call this function with a connected MongoClient.

```
await updateAllListingsToHavePropertyType(client);
```

Below is the output from executing the previous command.

```
3 document(s) matched the query criteria.
3 document(s) was/were updated.
```

Now our "Cozy Cottage" document and all of the other documents in the Airbnb collection have the `property_type` field.

```
{
_id: 5db9d9286c503eb624d036a1,
name: 'Cozy Cottage',
bathrooms: 1,
bedrooms: 2,
beds: 2,
property_type: 'Unknown'
}
```

Listings that contained a `property_type` before we called `updateMany()` remain as they were. For example, the "Spectacular Modern Uptown Duplex" listing still has `property_type` set to `Apartment`.

```
{
_id: '582364',
listing_url: 'https://www.airbnb.com/rooms/582364',
name: 'Spectacular Modern Uptown Duplex',
property_type: 'Apartment',
room_type: 'Entire home/apt',
bedrooms: 4,
beds: 7
...
}
```

If you're not a fan of copying and pasting, you can get a full copy of the code above in the Node.js Quick Start GitHub Repo.

Delete

Now that we know how to **create**, **read**, and **update** documents, let's tackle the final CRUD operation: **delete**.

Delete One Document

Let's begin by deleting a single Airbnb listing in the listingsAndReviews collection.

We can delete a single document by calling Collection's deleteOne(). `deleteOne()` has one required parameter: a filter of type object. The filter is used to select the document to delete. You can think of the filter as essentially the same as the query param we used in findOne() and the filter param we used in updateOne(). You can include zero properties in the filter to search for all documents in the collection, or you can include one or more properties to narrow your search.

`deleteOne()` also has an optional `options` param. See the deleteOne() docs for more information on these options.

`deleteOne()` will delete the first document that matches the given query. Even if more than one document matches the query, only one document will be deleted. If you do not specify a filter, the first document found in natural order will be deleted.

Let's say we want to delete an Airbnb listing with a particular name. We can use `deleteOne()` to achieve this. We'll include the name of the listing in the filter param. We can create a function to delete a listing with a particular name.

```
async function deleteListingByName(client, nameOfListing) {
const result = await
client.db("sample_airbnb").collection("listingsAndReviews")
.deleteOne({ name: nameOfListing });
console.log(`${result.deletedCount} document(s) was/were deleted.`);
}
```

Let's say we want to delete the Airbnb listing we created in an earlier section that has the name "Cozy Cottage." We can call `deleteListingsByName()` by passing a connected MongoClient and the name "Cozy Cottage."

```
await deleteListingByName(client, "Cozy Cottage");
```

Executing the command above results in the following output.

```
1 document(s) was/were deleted.
```

If you're not a fan of copying and pasting, you can get a full copy of the code above in the Node.js Quick Start GitHub Repo.

Deleting Multiple Documents

Sometimes you'll want to delete more than one document at a time. In this case, you can use Collection's deleteMany().

Like `deleteOne()`, `deleteMany()` requires that you pass a filter of type object. You can choose to include options of type object as well.

Let's say we want to remove documents that have not been updated recently. We can call `deleteMany()` with a filter that searches for documents that were scraped prior to a particular date. Our function will look like the following.

```
async function deleteListingsScrapedBeforeDate(client, date) {
const result = await client.db("sample_airbnb").collection("listingsAndReviews")
.deleteMany({ "last_scraped": { $lt: date } });
console.log(`${result.deletedCount} document(s) was/were deleted.`);
}
```

To delete listings that were scraped prior to February 15, 2019, we can call `deleteListingsScrapedBeforeDate()` with a connected MongoClient and a Date instance that represents February 15.

```
await deleteListingsScrapedBeforeDate(client, new Date("2019-02-15"));
```

Executing the command above will result in the following output.

```
606 document(s) was/were deleted.
```

Now only recently scraped documents are in our collection.