Your First MongoDB Tutorial

Hey gang, and welcome to your very first MongoDB tutorial!

Before we begin, let's quickly discuss what MongoDB is. If you're here, you probably already know it's a database for storing data (user data, blog posts, etc.). But let's delve deeper into its type and what sets it apart.

MongoDB is a NoSQL database—meaning we don't use SQL commands. This contrasts with SQL databases like MySQL.

SQL Databases (Relational Databases)

SQL databases, also known as relational databases (like MySQL), consist of tables with rows and columns. Each table stores a specific data type (e.g., users, blog posts). Each row is a record (a user record, a blog post record), and each column is a property of that record (username, email, ID, etc.). Often, different data tables relate to each other (e.g., a one-to-many relationship between authors and their books). SQL commands, like `SELECT * FROM authors`, query this data.

NoSQL Databases (MongoDB)

A NoSQL database like MongoDB differs significantly. Instead of tables, rows, and columns, it uses **collections** and **documents**.

- Each collection stores a specific data type (users, authors, books).
- Records within collections are called **documents**, resembling JSON objects with key-value pairs.

This structure offers several advantages:

- **Easier to use with JavaScript:** Documents are similar to JSON or JavaScript objects.
- **Nested documents:** A document can contain nested documents (e.g., an author document within a book document), providing flexibility compared to separate tables in SQL. You could also use separate collections, offering choices in data structuring.
- **Simpler querying (often):** MongoDB's simple methods make writing and reading data easy.
- **High-speed performance:** Queries are fast.

What You'll Learn

In this series, you'll learn to use MongoDB from scratch:

- Local installation and interaction using the MongoDB shell and Compass GUI.
- Basic operations: saving, retrieving, updating, and deleting data.
- Advanced concepts: filtering, sorting, and complex queries.

- Building a simple Node API interacting with MongoDB (CRUD operations).
- Testing API endpoints using Postman.
- Setting up a hosted database on MongoDB Atlas.

Prerequisites

Before starting, a basic understanding of a programming language and JSON is highly recommended. For the Node API section, basic Node knowledge is beneficial. Consider checking out my Modern JavaScript course and Node Crash Course (links below) if needed.

Accessing the Course

To watch this course ad-free, visit Netninja.dev. You can purchase the course for $2 or sign up for Netninja Pro ($9.99/month, first month half price with promo code – link below). Netninja Pro provides ad-free access to all courses, including premium content not on YouTube.

Don't forget to share, subscribe, and like! See you in the next lesson!

Setting Up MongoDB

Before we begin working with MongoDB, we need a database. There are two main options: using a hosted service like MongoDB Atlas, or installing MongoDB locally. This series focuses on the local installation method, but we'll cover MongoDB Atlas later.

Installing MongoDB Locally

1. **Download MongoDB:** Go to the MongoDB download page (link provided below). Download the free community server edition, selecting your desired version and operating system. I'm using the current version on Windows.
2. **Install MongoDB:** Double-click the installer and follow the wizard. Ensure the following options are selected:
    - "Install mongod as a service" (to easily start and stop the service)
    - "Install MongoDB Compass" (a graphical user interface for interacting with the database)
3. **Verify Installation:** Once installed, MongoDB Compass may automatically launch. Minimize it for now.

Installing the MongoDB Shell

The community server edition doesn't include the MongoDB shell, a command-line tool for interacting with the database. We'll use this extensively, especially in the early parts of the course to learn the basics.

1. **Download the Shell:** In the MongoDB tools section, download the MongoDB shell, selecting the same version and operating system as before.
2. **Install the Shell:** Install the shell by double-clicking the installer and following the prompts.
3. **Verify Shell Installation:** Open a terminal (command prompt, etc.) and type `mongo` then press Enter. If the installation was successful, you'll enter the interactive MongoDB shell, displaying the MongoDB version.

Using MongoDB Shell and Compass

We'll use both the MongoDB shell (command-line) and MongoDB Compass (GUI) throughout this series. The shell is particularly useful for learning fundamental concepts in an environment independent of any specific programming language.

Next Steps

In the next lesson, we'll explore how MongoDB stores data using collections and documents.

Data Structure in MongoDB

Before interacting with MongoDB, let's discuss its data structure. Understanding this will simplify database interaction.

MongoDB stores data in **collections**. A database can contain numerous collections for different data types. For example, a database might have three collections:

- Users
- Blog Posts
- Comments

The "users" collection would store user data (user documents). The "blog posts" collection would store blog post documents, and the "comments" collection would store comment documents. Different data types (documents) are grouped into their respective collections, simplifying retrieval of all documents from a single collection. For instance, fetching all blog post titles for display on a website involves retrieving all documents from the "blog posts" collection.

Documents

Documents represent individual records within a collection. The "blog posts" collection contains many blog post documents, each representing a single blog post.

A document's structure resembles a JSON object (key-value pairs), although it's stored as BSON (Binary JSON). For practical purposes, consider it a JSON object. When fetching documents, you receive JSON objects.

A blog post document might look like this:

```
{
  "title": "My Blog Post",
  "author": "John Doe",
  "tags": ["mongodb", "database"],
  "upvotes": 10,
  "body": "This is the content of my blog post."
}
```

Besides custom properties, each document has a unique `_id` property (a special ObjectID type in MongoDB). MongoDB assigns this ID upon document creation. You can query MongoDB using this ID to fetch a specific document. The JSON-like structure of documents simplifies data handling.

Nested Documents

Documents can have properties whose values are also documents or arrays of documents (nested documents). For example, the `author` property could be a nested document:

```
{
  "firstName": "John",
  "email": "john.doe@example.com",
  "role": "Author"
}
```

This is an alternative to referencing an author document in a separate collection. We'll explore nested documents and arrays later. For now, remember that nesting is possible.

This overview provides a foundational understanding of how data is structured in MongoDB collections and documents. Let's proceed to explore MongoDB Compass in the next lesson.

Getting Started with MongoDB Compass

Alright then, gang! Now that you know a little about how data is structured in MongoDB, let's start working with it on your computer. When you installed MongoDB in lesson two, the MongoDB Compass GUI tool was also installed. While you'll mostly interact with MongoDB directly from the shell or within your code, Compass is helpful for learning and visualizing your data. It shows your databases, collections, and documents, and lets you easily add, update, and delete documents.

This lesson provides a quick tour of the UI, how to connect to a database, and how to add data.

Connecting to MongoDB Compass

First, open Compass. The welcome screen should prompt you for a connection string. A connection string is a special MongoDB URL used to connect to a MongoDB cluster. If you use a service like MongoDB Atlas (for online, hosted databases), you'd paste its connection string here.

However, since we're working locally, simply click "Connect" without entering a connection string. This connects to your local MongoDB service and lists your databases. If this doesn't work, ensure your MongoDB service is running.

On Windows:

1. Search for "Services".
2. Find "MongoDB Server" in the Services window.
3. Ensure its status is "Running". If not, right-click and select "Start".

Exploring Existing and Creating New Databases

When you connect, you'll see existing databases. You might see some pre-made databases like "local" (containing startup log data). You can explore these by clicking on them and viewing their collections and documents. The left sidebar lists all databases, and the plus icon at the bottom lets you create a new one.

Let's create a database for a bookstore website:

1. Click the plus icon.
2. Name the database "Bookstore".
3. Compass will ask for your first collection. Let's call it "books".
4. Create the collection.

You'll now be in the "books" collection within the "Bookstore" database. Clicking the database name takes you to the database overview, showing all collections (you can create more with the "Create" button).

I'll delete my existing test databases ("MyDB" and "testing") to avoid confusion. To delete a database, click the trash icon next to it, confirm the database name, and click "Drop".

Adding Data to Your Collection

Now let's add data to the "books" collection.

To add a new document:

1. Click "Add Data" then "Insert Document".
2. Notice the automatically generated `_id` field; you can delete it, but MongoDB will add one anyway.
3. Create your document (similar to a JSON object):

```
{
  "title": "Name of the Wind",
  "author": "Patrick Rothfuss",
  "pages": 500,
  "genres": ["fantasy", "magical realism"],
  "rating": 9
}
```

4. Click "Insert".

You can add multiple documents at once by pasting them as an array of JSON objects. For example:

```
[
  { "title": "The Final Empire", ... },
  { "title": "The Way of Kings", ... },
  { "title": "The Call of the Wild", ... }
]
```

Managing Documents

Hovering over a document shows icons to delete (trash can) or edit (pencil) it. Editing lets you change values and update the document.

Filtering Data

You can filter data using a query object (more on this later). For example, to find all books with a rating of 9:

```
{ "rating": 9 }
```

Click "Find" to apply the filter.

This covers the basics of MongoDB Compass. We'll use it occasionally to visualize data. Next, we'll move on to the MongoDB shell.

Interacting with MongoDB using the Shell

So far, we've interacted with the MongoDB server using Compass, the GUI tool installed with MongoDB. Another way to interact with our database is via the MongoDB shell, where we execute commands to fetch, create, delete, and update data. This approach closely mirrors how we'll interact with MongoDB from our application code. For much of this course, I'll use the MongoDB shell for instruction. Everything we learn in the shell can be directly applied to building applications that use MongoDB (we'll cover that in the second half of the course).

Accessing the MongoDB Shell

To use the MongoDB shell, you can use the interactive shell bundled with Compass or any other terminal. To use it in a separate terminal, you'll need the MongoDB shell installed (see the first lesson in this series if you haven't already). The usage is the same regardless of the terminal.

In Compass, open the integrated terminal and start typing commands. For example, `show dbs` (short for databases) lists all current databases. You'll notice a "Test" database listed on the left; this indicates the current working database. Currently, "Test" doesn't exist.

We can switch to an existing database using the `use` command followed by the database name (e.g., `use Bookstore`). The left-hand indicator will update to reflect the current database. We can switch back to "Test" with `use test`. Note that the shell doesn't require a database to exist before we work with it. We can use a non-existent database (e.g., `use MyDB`) and MongoDB will create it when we add a collection and data.

For the remainder of this section, we'll use the MongoDB shell in a separate terminal. However, feel free to continue using the Compass-integrated shell.

To use the shell in another terminal, open your preferred terminal (I'm using Windows Terminal) and type `mongosh` then press Enter. This launches the interactive MongoDB shell, similar to the Compass version.

Basic MongoDB Shell Commands

The commands are largely the same in both shells. Here are some basic examples:

- **`show dbs`**: Lists all databases.
- **`use <database_name>`**: Switches to the specified database.
- **`cls`**: Clears the screen.
- **`db`**: Displays the current database.
- **`show collections`**: Lists all collections in the current database.
- **Variable Creation:** You can create and manipulate variables (e.g., `var name = "Yoshi";`).
- **`help`**: Displays a list of available commands.

- **`exit`**: Exits the MongoDB shell.

Let's demonstrate a few: `show dbs` lists our databases. `use bookstore` switches to the "bookstore" database. `cls` clears the screen for better readability. `db` shows the current database ("bookstore"). `show collections` lists collections within the database (in this case, "books").

This covers the basics of using the MongoDB shell. Next, we'll explore adding documents to a collection. I will be using the separate terminal moving forward, but you can continue using the Compass integrated shell if you prefer.

Adding Documents to a MongoDB Collection Using the Shell

Alright, then, gang. In the last couple of lessons, we learned how to use the MongoDB shell and Compass to interact with our databases. Going forward, we'll use both tools. Currently, in Compass, we have a `bookstore` database with a `books` collection containing several book documents (added in previous lessons). Let's learn to add documents from the shell.

First, ensure you're in the correct database. We're currently in the `test` database; let's switch to `bookstore`:

```
use bookstore
```

Now we're in the `bookstore` database. To add a document, we need to reference the collection:

```
db.books
```

This references the `books` collection (you can verify this by pressing Enter; it should show `bookstore.books`). Now, let's use the `insertOne` method to add a single document:

```
db.books.insertOne( {
    title: "The Color of Magic",
    author: "Terry Pratchett",
    pages: 300,
    rating: 7,
    genres: ["Fantasy", "Magic"]
} )
```

MongoDB automatically assigns a unique ID. This command adds a new document with the specified properties. Pressing Enter shows an acknowledgment with the newly created ID. Refreshing Compass will show the added document.

Adding Documents to a Non-Existent Collection

Interestingly, you don't need a pre-existing collection to insert a document. For example:

```
db.authors.insertOne( {
    name: "Brandon Sanderson",
    age: 60
} )
```

This creates the `authors` collection and inserts the document. You can verify this in Compass; the collection will be created, containing the new document. (I'll delete this collection afterward, using `db.authors.drop()`).

Inserting Multiple Documents

To insert multiple documents, use the `insertMany` method:

```
db.books.insertMany( [
    { title: "The Light Fantastic", author: "Terry Pratchett", pages:
320, rating: 8, genres: ["Fantasy", "Humor"] },
    { title: "Good Omens", author: "Terry Pratchett and Neil Gaiman",
pages: 400, rating: 9, genres: ["Fantasy", "Humor"] }
] )
```

This inserts both documents. Again, verify the addition in Compass.

That's how to add documents using `insertOne` for single documents and `insertMany` for multiple documents within an array. Enjoy!

Fetching Documents from a MongoDB Collection

Alright, so now we know how to add new documents to a collection, and we can see all of those documents using Compass. Often, you'll want to fetch documents from a collection to use them. For example, in an API, you might need an endpoint that returns a list of books. The handler function for that endpoint would communicate with MongoDB to fetch book documents from the `books` collection.

How do we do that? It's simple: use MongoDB's `find` method. To fetch all books from the `books` collection, you'd use:

```
db.books.find()
```

Pressing Enter sends this to the database. The MongoDB shell outputs the first 20 books it finds. If we have more than 20 books (and we might have 200 in the future!), typing the command again iterates and displays the next 20, and so on.

This behavior differs slightly from how the `find` method works in application code (we'll learn more about that later). For now, since we're unlikely to have more than 20 documents, it's not a concern in the shell.

Let's clear the console. The `find` method grabs all documents, up to the first 20, without additional criteria. We don't specify "find all books where the author is Terry Pratchett," for example. We simply fetch all available books.

However, we *can* use filters to refine results, similar to what we saw in Compass. A filter is an object:

```
db.books.find({ author: "Terry Pratchett" })
```

This finds all books where the author is Terry Pratchett. We're limiting the results, which is very useful.

We can filter by multiple criteria. To find books by Terry Pratchett with a rating of 7:

```
db.books.find({ author: "Terry Pratchett", rating: 7 })
```

This uses an object with multiple key-value pairs. Only books matching both conditions are returned.

Limiting Returned Fields

Let's clear the console again. Sometimes, you might not need all fields from a document. You might only want the title and author on a webpage. The `find` method accepts a second argument—an object specifying which fields to return:

```
db.books.find({ author: "Brandon Sanderson" }, { title: 1, author: 1 })
```

This still filters by author, but the second argument limits the returned fields to `title` and `author`. The `_id` field is always included; we don't need to specify it.

If you want all books but only specific fields:

```
db.books.find({}, { title: 1, author: 1 })
```

An empty object `{}` as the first argument acts as an empty filter, returning all documents. The second argument still controls which fields are returned.

The `findOne` Method

The `find` method returns multiple documents. To find a single document (e.g., by ID), use `findOne`:

```
db.books.findOne({ _id: ObjectId("...") }) // Replace "..." with the
actual ObjectId
```

Replace `"..."` with the actual `ObjectId`. You can use other properties besides `_id` to filter; if multiple matches exist, only the first one is returned.

This covers finding and retrieving documents using the MongoDB shell's `find` and `findOne` methods, including filtering and field selection. The way we handle the data returned by `find` will differ slightly in application code, a topic we'll address later. For now, remember that `find` gets all documents, and `findOne` gets a single document.

Chaining Methods in MongoDB

Alright then. So we know that when we send queries to MongoDB to fetch data, we use the `find` method or the `findOne` method to find whatever documents or document that we want. We can also tack on extra methods after the `find` method to perform additional tasks like sorting data and limiting the amount of documents we ultimately get back. This act of tacking on extra methods is called method chaining because we're essentially chaining those methods together one after another.

Let's start by using a method and chaining it on. This method is called `count`. It counts the number of documents we would get back from the query. If we run a `find` query without any filters, it might bring back seven documents. However, let's try using it with a filter: `{ author: "Brandon Sanderson" }`. Now the count is two, because only two books have this author. That's the `count` method – pretty simple, right?

The `limit` Method

Another method is the `limit` method. This lets us limit how many documents we get back. For example, `limit(3)` will return only three documents. We can chain this with other methods. For instance: `find().limit(3).count()` will first limit the results to three and then count those three.

The `sort` Method

The `sort` method lets us sort the results. We pass an object specifying the field to sort by and the order (1 for ascending, -1 for descending). For example, to sort by title in ascending order: `sort({ title: 1 })`. To sort in descending order, use `sort({ title: -1 })`.

Chaining Multiple Methods Together

What if we want to chain everything we've done so far? Let's sort by title in ascending order and then limit the results to three documents:

```
db.collection.find().sort({ title: 1 }).limit(3)
```

This first sorts the documents alphabetically by title and then limits the results to the top three. We can see the results are sorted alphabetically by title and limited to three. Awesome!

Nested Documents in MongoDB

Okay, there, my friends. So now we know how to add new documents to a collection and find documents using the `find` method. Next, let's discuss the structure of documents, particularly nested documents.

We've seen the general structure of a document, much like a JSON object: key-value pairs where keys are field names (in MongoDB terminology), and values can be various types—strings, Booleans, numbers, arrays, etc. Critically, a field's value can also be a nested document (another JSON object) or an array of nested documents.

For example, a `stock` field could contain a nested document outlining a book's stock details (e.g., `count` and `price` properties). Similarly, a `reviews` field could hold an array of nested documents, each representing a book review with `name` and `body` fields. Fetching a book document would then also retrieve its reviews, allowing you to display book details and reviews together on a page.

Benefits of Nested Documents

This approach of using nested (or embedded) documents offers performance benefits. An alternative would be separate `books` and `reviews` collections, with reviews referencing books via IDs. Fetching a book and its reviews would require two queries. Nested documents streamline this to a single query.

However, if a book has many reviews, storing *all* reviews within the book document might become inefficient. A hybrid approach is better: store only the latest reviews within the book document and fetch older reviews from a separate collection on demand (e.g., via a user-initiated action).

Adding Nested Review Documents

Before adding nested documents, I've deleted the existing books to start fresh. Let's add a new book with a `reviews` property containing nested documents. In the terminal:

```
db.books.insertOne({
  title: "The Way of Kings",
  author: "Brandon Sanderson",
  rating: 9,
  pages: 400,
  genres: ["fantasy"],
  reviews: [
    { name: "Yoshi", body: "Great book!" },
    { name: "Mario", body: "So good!" }
  ]
})
```

This adds a document with nested review documents. We can verify this in Compass.

To add multiple books, use `insertMany`:

```
db.books.insertMany([
  /* ... (array of book documents with nested reviews) ... */
]);
```

(Note: The code for multiple book insertions is omitted for brevity; it would involve an array of similar JSON objects as above). After refreshing Compass, you'll see the added books, each with its `reviews` property containing nested documents.

That's how to add nested documents to your MongoDB documents!

MongoDB Query Operators

So far, we've used the `find` method with filters that look for exact matches. For example, a filter for `rating: 7` finds only books with a rating of exactly 7. But what if we want to find books with a rating of 7 *or higher*, or perhaps those with a rating *less than* 4? This requires query operators.

In MongoDB, operators are denoted by a dollar sign (`$`). Let's explore some basic operators for queries involving values less than or greater than a specific number.

Greater Than and Less Than Operators

Let's say we want all books with a rating greater than 7. We can use:

```
db.books.find({ rating: { $gt: 7 } })
```

`$gt` stands for "greater than." This query returns books with ratings of 8, 9, 10, etc., but *not* 7. Changing the value to `$gt: 8` would only return books rated 9 and 10.

For "less than," we use `$lt`:

```
db.books.find({ rating: { $lt: 8 } })
```

This would retrieve only books rated 7.

We can also use:

- `$lte` (less than or equal to)
- `$gte` (greater than or equal to)

For example:

```
db.books.find({ rating: { $lte: 8 } }) // Returns books rated 7 and 8.
db.books.find({ rating: { $gte: 8 } }) // Returns books rated 8 and
higher.
```

Multiple Filters

We can combine multiple filters. For example, to find books by Patrick Rothfuss with a rating greater than 7:

```
db.books.find({ author: "Patrick Rothfuss", rating: { $gt: 7 } })
```

The OR Operator

The `$or` operator finds documents where at least one of the specified conditions is true. `$or` takes an array of filter objects. For example, to find books with a rating of 7 or 9:

```
db.books.find({ $or: [{ rating: 7 }, { rating: 9 }] })
```

This will return all books matching either condition. We could also use `$or` to find books with a rating of 7 or an author of Terry Pratchett:

```
db.books.find({ $or: [{ rating: 7 }, { author: "Terry Pratchett" }] })
```

Combining Operators

Let's combine different operators. To find books with less than 300 pages or more than 400 pages:

```
db.books.find({
  $or: [
    { pages: { $lt: 300 } },
    { pages: { $gt: 400 } }
  ]
})
```

This demonstrates the use of `$lt`, `$gt`, and `$or` together.

These are just a few of MongoDB's many query operators. The MongoDB documentation provides a comprehensive list, and we'll explore more operators throughout this course.

MongoDB `in` and `nin` Operators

Alright D gang, in the last video, we covered MongoDB operators for more complex queries: <, <=, >, >=, and `$or`. This video introduces the `$in` and `$nin` (not in) operators.

The `$in` Operator

We use the `$in` operator to check if a field's value exists within a specified array of values. For example, let's find all books with a rating of 7, 8, or 9. While you *could* use the `$or` keyword, `$in` provides a simpler solution.

Here's how it works:

```
db.books.find({ rating: { $in: [7, 8, 9] } })
```

This query says: "Find all books where the `rating` field is within this array of values (7, 8, and 9)." The results will include books with ratings of 7, 8, or 9.

We can compare this to using `$or`:

```
db.books.find({ $or: [ { rating: 7 }, { rating: 8 }, { rating: 9 } ] })
```

Both achieve the same result, but `$in` is more concise.

The `$nin` Operator

The `$nin` operator is the opposite of `$in`. It finds documents where the specified field's value is *not* in the given array.

Let's modify our query to find books with ratings *other than* 7, 8, and 9:

```
db.books.find({ rating: { $nin: [7, 8, 9] } })
```

This will return only books whose rating is not 7, 8, or 9. For example, if you remove 9 from the array, books with a rating of 9 will now be included in the results.

Summary

The `$in` and `$nin` operators provide efficient ways to query documents based on whether a field's value is present within (or absent from) a specified array of values. They offer a more concise alternative to using multiple `$or` conditions for checking against multiple values.

Querying Arrays and Nested Documents

Alright then, gang. Let's examine the structure of our documents. We have several properties or fields: some are strings, some are integers, and we have two array values. The `genres` array is a simple array of strings, while the `reviews` array contains nested documents. Each nested document in `reviews` has `name` and `body` properties.

This lesson focuses on querying documents based on these array values. Previously, our queries used single-value properties like `title`, `author`, `pages`, and `rating`. Querying arrays requires a slightly different approach.

Querying Simple Arrays: The `genres` Array

Let's start with the simpler `genres` array. For example, we might want to fetch all books with the genre "fantasy" or "sci-fi". How do we do this?

In the shell, we use `db.books.find()`:

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

This means: "Look in the `genres` property. If this array contains 'fantasy', return the book." Note the difference from querying single-value fields. When querying `author`, we'd need an exact match. Here, MongoDB checks for the element's *existence* within the array.

Let's try it: The output will show all books with "fantasy" in their `genres` array (even if other genres are also present). Changing "fantasy" to "magic" will return books containing "magic".

Exact Array Matches

What if we need an *exact* match? To find books where "magic" is the *only* genre, we specify the array value directly:

```
db.books.find({ genres: ["magic"] })
```

This returns no results because no book in our collection has "magic" as its sole genre. However, changing this to `genres: ["fantasy"]` might return books where "fantasy" is the only genre. You can also specify multiple values for an exact match: `genres: ["fantasy", "magic"]` would return books with *only* "fantasy" and "magic" as genres.

Querying Arrays with Multiple Conditions: The `$all` Operator

What if we want books containing both "fantasy" and "sci-fi" in their `genres` array, regardless of other genres present? We can't use individual queries because that would return books with either "fantasy" *or* "sci-fi." Instead, we use the `$all` operator:

```
db.books.find({ genres: { $all: ["fantasy", "sci-fi"] } })
```

This checks if *all* elements within the specified array (`["fantasy", "sci-fi"]`) are present in the `genres` array of each document.

Querying Nested Documents: The `reviews` Array

Finally, let's query based on our nested documents within the `reviews` array. Each review has a `name` property. Let's find all books with a review by "Luigi":

```
db.books.find({ "reviews.name": "Luigi" })
```

We use dot notation ("reviews.name") to access the `name` field within the `reviews` array. The property name ("reviews.name") is enclosed in quotes because we are using dot notation. This query returns all books containing a review with the name "Luigi".

That's it for querying arrays and nested documents! Remember the differences between simple existence checks, exact matches, using the `$all` operator, and querying nested documents using dot notation.

Deleting Documents from a Collection

So now we've seen how to add and find documents. Let's switch our focus to deleting them. Before we begin, let's export our current data as a backup.

Exporting Your Collection Data

This is useful because we'll be deleting data, and it's easier than manually recreating everything. In Compass, click the "Export collection" button. Export the full collection, selecting all fields. Choose JSON as the output format, browse to your desired save location, and click "Export". I've already done this; the file is on my desktop. Later, we can import this JSON file via "Add data" > "Import file".

Deleting Documents

Like adding documents, there are two main methods: `deleteOne` and `deleteMany`.

`deleteOne`

Let's start with `deleteOne`, which is simpler. First, let's see our current books: `db.books.find()`.

Now, we'll grab the ID of a book to delete. We'll use the `_id` property because it's unique. (You *could* use another field like "author," but that would only delete the *first* matching document.)

Copy the `_id` and use it in the following command:

```
db.books.deleteOne({ _id: ObjectId("YOUR_OBJECT_ID_HERE") })
```

Replace `"YOUR_OBJECT_ID_HERE"` with the actual `_id`. After pressing Enter, check `acknowledged` (should be `true`) and `deletedCount` (should be `1`). Then, use `db.books.find()` again to confirm the book is gone.

`deleteMany`

The `deleteMany` method allows deleting multiple documents at once. Let's delete all books by Terry Pratchett:

```
db.books.deleteMany({ author: "Terry Pratchett" })
```

This will delete all documents where the `author` field is "Terry Pratchett". Check `acknowledged` and `deletedCount` (should be `2` in this case). Use `db.books.find()` to verify the deletion.

Importing Your Data

Now, our collection is empty. Let's import our saved backup. Go to "Add data" > "Import file", select the JSON file (`books.json` in my case), and import it. You should now see all your documents restored.

Updating Documents in MongoDB

Alright there, my friends! So we've seen how to add, find, and delete documents. Now, let's learn how to update them using the `update()` methods.

First, let's list all the books using the `find()` method:

```
DB.Books.find()
```

Let's say we want to update this book: We'll update the `Pages` to 350 and the `rating` to 8. We'll use the `_id` to specify the document:

To update a single book, we use the `updateOne()` method. First, copy the `_id`. Then:

```
db.Books.updateOne(
  { _id: ObjectId("YOUR_OBJECT_ID") }, // Replace YOUR_OBJECT_ID
  { $set: { rating: 8, pages: 360 } }
)
```

This takes two arguments:

1. An object specifying the document to update (based on the `_id` for uniqueness). You *could* use another field, but if it's not unique, only the first matching document will be updated.
2. An object specifying the fields to update. We use the `$set` operator to set new values for multiple fields.

After running this, check the `modifiedCount`: A value of `1` confirms one document was updated. Let's verify with:

```
db.Books.find()
```

Updating Multiple Documents

Now, let's say we want to update multiple documents at once. Imagine a spelling error in "Terry Pratchett's" author name across multiple entries. We'll use `updateMany()`:

```
db.Books.updateMany(
  { author: "Terry Pratchett" },
  { $set: { author: "Terry Pratchett" } } //Corrected spelling
)
```

This updates all documents where `author` is "Terry Pratchett". The `modifiedCount` will reflect how many were updated. Again, verify with `db.Books.find()`.

MongoDB Operators

Let's explore some useful MongoDB operators:

`$inc` Operator

The `$inc` operator increments or decrements a numeric field. Let's increase the page count of a book by 2:

```
db.books.updateOne(
  { _id: ObjectId("YOUR_OBJECT_ID") },  // Replace YOUR_OBJECT_ID
  { $inc: { pages: 2 } }
)
```

To decrement, use a negative value: `{ $inc: { pages: -2 } }`

`$push` and `$pull` Operators

These operators manage array elements:

- `$push`: Adds an element to an array.
- `$pull`: Removes an element from an array.

Let's remove "fantasy" from a genre array and then add it back:

```
//Remove
db.books.updateOne(
  { _id: ObjectId("YOUR_OBJECT_ID") },  // Replace YOUR_OBJECT_ID
  { $pull: { genres: "fantasy" } }
)

//Add back
db.books.updateOne(
  { _id: ObjectId("YOUR_OBJECT_ID") },  // Replace YOUR_OBJECT_ID
  { $push: { genres: "fantasy" } }
)
```

`$each` Operator

The `$each` operator allows pushing multiple elements into an array at once:

```
db.books.updateOne(
  { _id: ObjectId("YOUR_OBJECT_ID") },  // Replace YOUR_OBJECT_ID
  { $push: { genres: { $each: ["one", "two"] } } }
)
```

This adds "one" and "two" to the `genres` array.

There are several ways to update documents in MongoDB; these examples provide a solid foundation.

Connecting a Node.js Application to MongoDB

Alright, so far we've explored MongoDB interactions directly using the shell and Compass. However, most real-world applications interact with MongoDB programmatically, from within application code (like Node.js, Python, etc.). This requires using MongoDB drivers, which provide language-specific bindings. For a Python application, you'd use the Python driver; for Node.js, the Node.js driver, and so on. These drivers facilitate programmatic communication with MongoDB.

You can find a comprehensive list of drivers for various programming languages on the MongoDB website under "Resources" in the top navigation, then selecting "Drivers."

The remainder of this series focuses on building a Node.js API that interacts with MongoDB, utilizing the Node.js driver to bridge our application and the database. The following sections detail driver installation and usage.

Prerequisites

Before proceeding, ensure you have the following:

- **Node.js installed:** This is essential for following along.
- **Basic Node.js knowledge:** A foundational understanding of Node.js applications and creating simple Express apps is recommended. If you're new to Node.js, I highly recommend checking out my Node.js crash course (link below).

Setting Up the Node.js Application

1. **Create a new project folder:** Open a new folder in your preferred text editor (I'm using VS Code).

2. **Initialize the project:** Open an integrated terminal (in VS Code: `Terminal > New Terminal`; otherwise, use Command Prompt, Windows Terminal, or your preferred terminal). Navigate to your project directory using the `cd` command. Then, run `npm init`. Accept the default options to create a `package.json` file, which tracks project dependencies.

3. **Create an entry file:** Create a new file named `app.js`. This will contain the majority of our application code.

4. **Install Express.js:** We'll use Express.js for our API. Run `npm install express --save` to install and save it as a dependency. You should now see `express` listed in your `package.json` file.

5. **Create the Express app in `app.js`:**

```
// init app & middleware
const express = require('express');
const app = express();

// Listen for requests on port 3000
app.listen(3000, () => {
  console.log('App listening on port 3000');
});

// Routes
app.get('/books', (req, res) => {
  res.json({ message: 'Welcome to the API' });
});
```

This code initializes an Express app, listens on port 3000, and sets up a GET request handler for `/books`. For now, it sends a simple JSON response. If any of this is unclear, please refer to my Node.js crash course.

6. **Run the application:** Install Nodemon globally (`npm install -g nodemon`) for automatic server restarts on file changes. Then, run `nodemon app.js`. You should see "App listening on port 3000" in your console. Access `http://localhost:3000/books` in your browser to see the JSON response.

Installing the MongoDB Driver

1. **Open a new terminal.** Ensure you're in your project directory.
2. **Install the MongoDB driver:** Run `npm install mongodb --save`. This installs the MongoDB Node.js driver and adds it to your dependencies in `package.json`.

We'll use this driver in the next lesson to connect to our MongoDB database.

Connecting to MongoDB

Alright then gang. So in the last lesson, we set up an Express app and installed the Node MongoDB driver. The next thing we want to do is connect to MongoDB using that driver so we can interact with it to fetch and add data.

To do this, I'm going to make a new file called `db.js` (database.js). You don't have to do this; you can keep all your code in the `app.js` file, but I like to keep my code clean, reusable, and modular. Therefore, I'm putting all my database connection code in a separate file.

Ultimately, this file will contain two functions:

1. One to initially connect to the database.
2. One to retrieve the database connection after it's established.

Both functions will be exported so we can use them in our `app.js` file.

Let's start by exporting an object:

```
module.exports = {
  // ... functions will go here
};
```

This is how we export things in a Node application. Inside this object, we'll have two properties representing our functions:

- `connectToDb`: This function will initially connect to the database.
- `getDb`: This function will return the database connection after it's established.

The process involves two steps:

1. Establish a connection using `connectToDb`.
2. Retrieve the connection using `getDb`.

Next, we need to import `MongoClient` from the MongoDB driver:

```
const { MongoClient } = require('mongodb');
```

This destructures the `MongoClient` object. We'll use it inside the `connectToDb` function to connect to the database:

```
MongoClient.connect(connectionString, { /* options */ })
  .then(client => { /* ... */ })
  .catch(err => { /* ... */ });
```

The `connect` method takes a connection string as an argument. This is a special MongoDB URL. When working locally, the connection string looks like this:

```
mongodb://localhost:27017/bookstore
```

Replace `bookstore` with your database name. Later, I'll show you how to connect to a remote database using MongoDB Atlas.

The `connect` method is an asynchronous task that returns a promise. We use `.then()` to handle the successful connection and `.catch()` to handle errors:

```
MongoClient.connect(connectionString)
  .then((client) => {
    const dbConnection = client.db('bookstore'); // Get the database object
    // ... more code here ...
  })
  .catch((error) => {
    console.error(error); // Log any errors
  });
```

We'll initialize a `dbConnection` variable and update it after a successful connection.

The `getDb` Function

The `getDb` function's sole purpose is to return the `dbConnection`:

```
const getDb = () => dbConnection;
```

This can be a single-line function. We'll call this after connecting to the database.

Adding a Callback

Let's add a callback argument (`cb`) to the `connectToDb` function. This allows us to execute a function after the connection attempt (success or failure). We'll invoke the callback after updating `dbConnection` (success) or in the `catch` block (error), passing the error object as an argument:

```
const connectToDb = (cb) => {
  MongoClient.connect(connectionString)
    .then((client) => {
      dbConnection = client.db('bookstore');
      cb(); // Invoke callback on success
    })
    .catch((error) => {
      cb(error); // Invoke callback with error on failure
    });
};
```

This completes the `db.js` file. Now, let's call these functions from `app.js`.

Importing and Using Functions in `app.js`

First, import the functions:

```
const { connectToDb, getDb } = require('./db');
```

Ideally, we want to connect to the database before listening for requests:

```
// Database connection
connectToDb((error) => {
  if (!error) {
    app.listen(port, () => console.log(`Server listening on port
${port}`));
    db = getDb(); // Get the database connection object
  } else {
    console.error('Failed to connect to the database:', error);
  }
});
```

We pass an arrow function as a callback to `connectToDb`. If the connection is successful (`error` is null), we start listening for requests and get the `db` object using `getDb`. Otherwise, we log the error. Now we have the `db` object to interact with the database in our endpoint handler functions.

In the next lesson, we'll start fetching documents in our GET request handler.

Fetching All Books

Alright D gang, so now we've connected to the database, and we can communicate with it using the `DB` variable. We'll use this inside the `get` request handler function to find all documents in the `books` collection.

To do this, we can say `DB.collection()`. The `collection()` method references a specific collection; we pass in the collection name as a string. In our case, that's "books". This is different from the shell where we use `DB.books`; in Node.js and JavaScript, we use the `collection()` method.

After that, we use the `find()` method. However, before proceeding, let's discuss how `find()` works in a real application.

In the shell, `find()` seemingly returns all documents. Actually, it returns a *cursor*, an object pointing to a set of documents defined by our query. Without arguments, it points to the entire collection; with a filter, it points to a subset. The cursor exposes methods to fetch the data. Two useful methods are `toArray()` and `forEach()`.

- `toArray()`: Fetches all documents and puts them into an array.
- `forEach()`: Iterates through documents one at a time.

MongoDB fetches documents in batches (defaulting to 101) to manage network bandwidth, especially with large collections. If we use `forEach()`, it fetches and processes one batch at a time.

Remember, `find()` returns a cursor; we need cursor methods to access the documents. The shell automatically iterates over the first 20 documents and allows viewing more using `it`. This behavior is unique to the shell; in applications, we manually work with the cursor object.

Now, let's continue. We don't need a filter (we want all books). Before using a cursor method, let's use the `sort()` method (which also returns a cursor). We'll sort alphabetically by author:

```
.sort({ author: 1 })
```

Next, we'll use `forEach()` to iterate through each book:

```
let books = []; // Initialize an empty array

// ... find().sort().forEach(...) ...

.forEach(book => {
  books.push(book);
});
```

This is asynchronous because it fetches documents in batches. We use a `.then()` method to execute a function once the iteration completes:

```
.then(() => {
  response.status(200).json(books);
})
.catch(error => {
  response.status(500).json({ error: "Could not fetch documents" });
});
```

This sends a 200 OK response with the `books` array as JSON. The `.catch()` handles errors, sending a 500 Server Error response with an error message.

In summary: We initialize an empty `books` array, reference the "books" collection, find and sort the books. The `forEach()` method iterates through the books (in batches) adding each to the `books` array. Once complete, we send a 200 OK response with the `books` array. Errors result in a 500 Server Error response. Go to `localhost:3000/books` in your browser to see the JSON response.

In the next lesson, we'll fetch a single book using a URL like `/books/:id`.

Fetching a Single Book

In the previous video, we created a route handler to retrieve all books. Now, let's create a route for fetching a single book using its ID.

Creating the Route

The route will be a GET request to `/books/:id`, where `:id` is a route parameter representing the book's ID. In Express.js, route parameters are denoted with a colon (`:`) before the parameter name.

```
app.get('/books/:id', (req, res) => {
  // Handler function
});
```

Accessing the Route Parameter

Inside the handler function, we access the route parameter using `req.params.id`. For example, if the URL is `/books/123`, `req.params.id` will be `123`.

```
const id = req.params.id;
```

Fetching the Book from MongoDB

We'll use MongoDB's `findOne` method to retrieve the book from the database. The `_id` field in MongoDB documents needs a special `ObjectId` type.

```
const { ObjectId } = require('mongodb');

db.collection('books').findOne({ _id: new ObjectId(req.params.id) })
  .then(document => {
    res.status(200).json(document);
  })
  .catch(err => {
    res.status(500).json({ error: 'Could not fetch the document' });
  });
```

Remember to require the `mongodb` package to use the `ObjectId` constructor.

Handling Invalid IDs

The `ObjectId` constructor throws an error if the provided ID string is not properly formatted (12 bytes or 24 hex characters). To prevent this, we should validate the ID before creating the `ObjectId`. The `ObjectId.isValid()` method helps with this.

```
if (ObjectId.isValid(req.params.id)) {
  db.collection('books').findOne({ _id: new ObjectId(req.params.id) })
    .then(document => {
      res.status(200).json(document);
    })
    .catch(err => {
      res.status(500).json({ error: 'Could not fetch the document' });
    });
} else {
  res.status(500).json({ error: 'Not a valid document ID' });
}
```

This improved code first checks if the ID is valid. If not, it sends a custom error response. If valid, it proceeds to fetch the document.

Handling Non-Existent Documents

If a document with a valid ID doesn't exist, MongoDB returns `null`. You can handle this either on the server or client-side.

Testing

Testing with valid and invalid IDs in the browser confirms the functionality. A properly formatted but non-existent ID returns `null`, while an improperly formatted ID returns the custom error message.

Next Steps

In the next lesson, we'll transition from using the browser for testing to using Postman.

Introducing Postman for API Testing

Alright then gang. So far, so good! We've created two endpoints: one to fetch all documents and another to fetch single documents. Both are working. Currently, we're testing these endpoints in the browser, which is fine for now. However, moving forward, we'll use Postman.

Postman simulates requests to an API (which is what we're building!), showing us the responses. The main reason for using Postman instead of the browser is that we'll soon be making DELETE, POST, and UPDATE requests. These are difficult to do from the browser without writing front-end JavaScript – something we want to avoid to stay focused on MongoDB. Postman simplifies these requests and lets us save and organize them.

Getting Started with Postman

You can download Postman for free at postman.com/downloads (postman.com/downloads). Click the download button and follow the installation instructions. You might also need to sign up for a free account.

Once installed and signed up, open Postman. The screen should look something like this (minor differences may exist depending on your version). We won't deep-dive into every Postman feature, but we'll cover the basics of making and saving requests.

Making and Saving GET Requests

Let's start with a GET request for all book documents. Create a new request by clicking the plus icon or going to `File > New Tab`.

1. **Select the request type:** In the dropdown, choose "GET".
2. **Enter the URL:** Use `http://localhost:3000/books`. This is the same endpoint we've been using in the browser.
3. **Send the request:** Click the "Send" button.

You should see a JSON response containing your book documents at the bottom. If so, it worked!

Now, let's save this request:

1. Click the "Save" button.
2. You can add a title (or use the URL).
3. Create a new collection (I'll call mine "Bookstore").
4. Click "Save".

Your saved request will appear in the "Bookstore" collection on the left. You can easily reopen and resend it later.

Making a GET Request for a Single Book

Let's create a request for a single book.

1. Click the plus icon for a new tab.
2. Select "GET".
3. Copy and paste the following URL (replacing the ID with an actual book ID): `http://localhost:3000/books/[bookID]` *Example: `http://localhost:3000/books/654321`*
4. Click "Send".

You should see the details of that specific book. Save this request to the "Bookstore" collection as well.

Now we have two saved requests. In future lessons, we'll create POST, PATCH, and DELETE requests to test other endpoints. For now, we have Postman installed and configured. The next lesson will cover creating a request handler for POST requests and testing it with Postman.

Setting Up a POST Request Handler

Alright, then. So now we know we can make POST requests from Postman (the tool I showed you in the last lecture). What I'd like to do now is set up a POST request handler inside our `app.js` file so we can test the endpoint.

Down here, I'm going to say `app`, and then it's a POST request this time that we want to handle:

```
app.post('/books', (req, res) => {
  // ...
});
```

We're sending a POST request to `localhost:3000/books`. When we receive that POST request, we fire a function that receives `request` and `response` arguments.

Inside this function, the first thing we want to do is get the body of the POST request. When we send a POST request, we send a request body containing the information we want to save to the database (in this case, a book document). We get the request body using the `req.body` property, but we need middleware provided by Express:

```
app.use(express.json());
```

This `app.use(express.json())` passes any incoming JSON so we can use it in our handler functions. Now we can access the request body:

```
const book = req.body;
```

Now we have the `book` object. We want to save it to the database using the `insertOne` method:

```
db.collection('books').insertOne(book)
  .then(result => {
    res.status(201).json(result);
  })
  .catch(error => {
    res.status(500).json({ error: 'Could not create a new document' });
  });
```

This asynchronously inserts the document. The `.then` method handles a successful insertion, sending a 201 status code and the result from MongoDB. The `.catch` block handles errors, sending a 500 status code and an error message.

Testing the POST Request in Postman

Now let's test this POST request in Postman. Create a new request, change the type to POST, and set the URL to `http://localhost:3000/books`.

Since we're saving a new document, we need a request body. Go to the "Body" tab, select "raw", and choose JSON. Here's the book object I'll use:

```
{
  "title": "The Final Empire",
  "author": "Brandon Sanderson",
  "rating": 4.5,
  "pages": 560,
  "genres": ["Fantasy", "Epic"],
  "review": {
    "reviewer": "John Doe",
    "text": "A great read!"
  }
}
```

Click "Send." If successful, you'll see an acknowledgment and the inserted ID. Let's verify this by retrieving all books and checking for our new entry. We can see "The Final Empire" with a matching ID. Awesome!

Finally, let's save this Postman request for later use. I'll name it "Bookstore" and save it.

Now we have GET requests for all books and a single book, and a POST request to add new books. Next up, we'll look at DELETE requests.

Creating a DELETE Request to Delete Books

All right, then, so now we have our GET requests for all books and a single book, as well as a POST request to add new books. The next thing I'd like to do is create a DELETE request to delete a specific book. Let's do that.

In our handler, we'll use `app.delete`. The endpoint will be `/books/:id`, where `:id` is the ID of the book we want to delete. The function will take the request and response objects. Inside that function, we need to delete a book from the database. This will be similar to the `findOne` method.

First, we'll check if the object ID is valid. We don't want to try deleting an invalid document, as this will cause an error. Then, we'll get the books collection, use the `deleteOne` method, and return a response indicating success or failure. If the ID is invalid, we'll send a different response.

Instead of writing this from scratch, I'll copy and modify the existing `findOne` code:

1. Change `findOne` to `deleteOne`.
2. We receive a result object instead of a document, which we'll send back to the client within the JSON method.
3. We'll catch any errors, set a 500 status code, and return a "Could not delete the document" message.
4. The response for an invalid object ID remains the same.

So, we pass in the ID, check if it's a valid object ID, get the books collection, use `deleteOne` specifying the book ID from the route parameter, send the result back to the client, and handle any errors.

Testing the DELETE Request in Postman

Let's create a new DELETE request to `localhost:3000/books/:id`, replacing `:id` with the ID of the book we want to delete. I'll get an ID from the GET request for all books:

1. Send the GET request.
2. Find a book ID (e.g., the George Orwell book).
3. Copy the ID.
4. Paste the ID into the DELETE request.
5. Send the request.

We should receive a response from the server. The response will show `acknowledged: true` and `deletedCount: 1`, confirming the deletion. Verifying in the GET request for all books will show the book is no longer present.

Awesome! That worked. Now we have GET requests for all books and single books, a POST request to add books, and a DELETE request to delete books. I'll save this in the `bookstore` folder.

Next Steps: PATCH Request

Next up, we'll handle a PATCH request to update existing books in the database.

Updating Existing Books with PATCH Requests

Okay then, so there's one more request type to handle: updating an existing book in the database. We might want to change the title, author, or other fields. To do this, let's create a PATCH request.

We use a PATCH request to update individual fields in a document. It can be one field or many—it doesn't matter. The endpoint will be `/books/{id}`, where `{id}` is the ID of the book to update (a route parameter).

When we receive this request, a function is fired, taking the request and response objects as input. Inside this function:

1. **Get the Updates:** Similar to POST requests, we'll receive a request body containing the new fields and values. We'll grab this and call it `updates`:

   ```
   const updates = request.body;
   ```

   `updates` will be an object. We don't need to update all fields; we can update just one or two (e.g., `author` and `title`, or `rating` and `pages`).

2. **Validate the ID:** We'll perform the same check as with the DELETE request to ensure the `ObjectId` is valid. This involves checking if the ID is present and if it's a valid MongoDB ObjectId.

3. **Update the Document:** We'll use MongoDB's `updateOne` method:

   ```
   booksCollection.updateOne(
       { _id: ObjectId(id) },
       { $set: updates }
   );
   ```

   The first argument specifies the query to find the book (based on the ID). The second argument uses the `$set` operator to update the specified fields. Any fields not included in `updates` remain unchanged.

   The `updates` object might look like this:

   ```
   {
       "title": "New Title",
       "rating": 8
   }
   ```

4. **Handle the Response:** The `.then()` method handles the successful update, sending a 200 OK response with the result from MongoDB. A `.catch()` block handles errors, sending a 500 Internal Server Error response.

Testing the PATCH Request with Postman

Let's test this with Postman. We'll use a PATCH request to `/books/{id}`.

1. **Choose a Book:** Use a GET request to `/books` to get a list of books and select one to update.
2. **Send the PATCH Request:** In Postman, create a PATCH request to the selected book's URL. Send a JSON body with the fields you want to update:

   ```
   {
       "pages": 350,
       "rating": 8
   }
   ```

   You can update as many or as few fields as needed.
3. **Check the Results:** The response will indicate success (e.g., `acknowledged: true`, `modifiedCount: 1`). Verify the update by using another GET request to `/books`.

Next Steps: Indexes and Pagination

We've covered the basic CRUD operations. In the next lesson, we'll discuss indexes and pagination to improve our API's performance and scalability.

Implementing Pagination in Your API

All right then, gang. So now we've made all the basic CRUD endpoints for the API to create, read, update, and delete data from the database. Awesome! Now there's one more thing I want to do for this API: implement simple pagination for the GET request for all the books.

If we had a thousand books, we probably wouldn't want to get all of them back at once as a response. It's much better to get a certain amount at a time, like 20, for example. We could list those 20 books on a page, and if a user wanted to see more, they could click a button to request more. We'd then ask MongoDB for the next 20, and so forth.

That's what we're going to achieve in this lesson, but *without* the UI, front end, and button. We'll do this using query parameters in the endpoint URL.

For example, to get the first page of book documents, we can add a query parameter to the endpoint for all books. This parameter could be called `pages`, or `p` for short. We'd set that query parameter equal to a page number. I'm going to start at 0, which gets the first page of results. To get the next page, we'd change that value to 1, then 2, and so on. This is how the request will look.

But how do we handle that in the API on the backend? It's pretty simple. To get a query parameter value in an Express application, we access it from the `request` object.

I'll create a constant called `page`, and set it equal to `request.query.p` (or `request.query.page` if you named your parameter differently). This gets us the value of the requested page.

It might be the case that the request doesn't include a page query parameter. In that case, I want it to default to 0 to get the first page of results. To do that, I can add a logical OR (two pipe symbols):

```
const page = request.query.p || 0;
```

If `request.query.p` doesn't have a value, the value of the `page` constant will be 0. If the parameter *does* have a value, the logical OR is ignored, and `page` will take the parameter's value.

Okay, so now we have the page of results we want to fetch. What next? We need to decide how many books per page should be sent back. This could be 10, 20, 30, or whatever you decide. For this tutorial, I'll set it to 3 because we don't have many documents, and I want to demonstrate several pages. In your application, you'll likely use a larger number.

Let's create a variable to store the books per page:

```
const booksPerPage = 3;
```

Next, we need to implement this pagination in the query sent to MongoDB. We can use a combination of two methods: `skip` and `limit`.

Imagine we have three books per page and want to get the second page (`page` = 1). We'd want to skip the first three books. We use the `skip` method to skip the number of books we want to skip: `page * booksPerPage`. If `page` is 1, we skip three books. Then, we use the `limit` method to fetch the next three (the next page of books).

Let's implement these methods:

```
const skipBooks = page * booksPerPage;

// ... MongoDB query ...
.skip(skipBooks)
.limit(booksPerPage);
```

When `page` is 0, `skipBooks` is 0, so we skip none and get the first three books.

Let's save this and test it. If we send a GET request without a query parameter, the `p` variable defaults to 0, so we should only see three results. If we set `p` to 1, it skips the first three and gets the next three (or fewer, if there aren't enough remaining). This is much better than fetching all documents at once, especially when dealing with a large number of records. The user can then click between pages, and you only fetch the data when a specific page is requested.

Indexes in MongoDB

Alright then, gang. So now we've pretty much completed the API, but there's one more topic I want to introduce: indexes. Indexes allow the database server to perform specific queries much more efficiently without examining the entire collection of documents.

Imagine sending a query to MongoDB to fetch all books with a rating of 10. Normally, MongoDB would scan the entire collection, examining each document's rating field. This is inefficient with thousands of documents. To combat this, we create an index.

An index for a database collection is like a book index: it lists what you need to look up and tells you where to find it. In MongoDB, an index is a list of values from a specific field in your documents. For example, an index based on the `rating` field would list each rating and a pointer to the corresponding document. When a query arrives (e.g., fetch all documents with a rating of 10), MongoDB only needs to check the index, quickly finding the relevant documents. This is much more efficient.

However, you don't always need indexes. Creating an index means updating it whenever the collection changes. More indexes mean more work during updates. Only use indexes for:

- Specific queries returning a subset of documents.
- Collections with many documents requiring sorting during queries.

Creating and Using Indexes

Let's see how this works. I'll show you a simple query to find books with a rating of 8. Currently, there are only five or six documents. Using the `explain()` method with the `executionStats` argument shows the query's execution statistics:

```
db.books.find({ rating: 8 }).explain("executionStats")
```

This reveals that all five documents were examined to find the two matching documents. Now, let's create an index on the `rating` field:

```
db.books.createIndex( { rating: 1 } )
```

This creates an index. We can list all indexes using:

```
db.books.getIndexes()
```

You'll see two indexes: one automatically created by MongoDB (based on the `_id` field), and the one we just created. Now, let's run the same query with `explain()` again:

```
db.books.find({ rating: 8 }).explain("executionStats")
```

This time, only two documents were examined – those matching the index. This demonstrates the efficiency gain. While this example is small (only five documents), it illustrates the principle.

Dropping Indexes

Finally, let's drop (delete) the index:

```
db.books.dropIndex("rating_1")
```

(Note: The exact index name might vary slightly depending on your MongoDB version). Verifying with `db.books.getIndexes()` shows only the `_id` index remains.

There you go, my friends! That's indexes in a nutshell.

Connecting Your Node & Express API to MongoDB Atlas

Alright there, my friends! So we've built a Node and Express API that communicates with MongoDB locally. For a production application, however, your app won't use your local MongoDB service. Instead, it'll connect to an online MongoDB server. One excellent way to do this is using MongoDB Atlas, a Database-as-a-Service platform. It lets you set up a free-tier cloud database easily and connect it to your application in minutes.

Getting Started with MongoDB Atlas

First, head to MongoDB/Atlas (https://www.mongodb.com/atlas) and sign up for a free account (or click the "Try Free" button). You can sign up directly or use Google (I'll use Google since I already have an account).

Once signed in, you'll be taken to your dashboard where you can create databases. It'll look something like this:

1. Click "Build a Database."
2. Choose the "Shared" option (it's free).
3. Click "Create."

You'll be asked a few questions:

- **Cloud Provider:** I'll stick with the default (AWS).
- **Region:** You can change this if needed; I'll use the default.
- **Cluster Name:** You can customize this; I'll use the default.

Click "Create" to create your cluster. This takes a few minutes. While it's creating, let's create a username and password to connect to the database from your code.

Setting Up Database Users and Network Access

I've already created a username and password. You can see them under "Database Access." I'm using the username "Yoshi" and password "test123". You can add a new user by filling in the form and clicking "Add," or by going to "Database Access" and creating one there.

Next, go to "Network Access" and allow access to your database. You can add your IP address for access from your computer, or, for testing purposes, allow access from anywhere (**don't do this in production!**). I'll allow access from anywhere for this example. Confirm the changes.

After a couple of minutes, the database should be created. Click the "Connect" button to get your connection string. You'll need to replace the placeholder username and password with your own.

Updating Your Code with the Connection String

Copy the connection string and go back to your code (the `db` file). Replace your existing connection string with the new one. Since it's long, I'll create a new variable:

```
const uri = "YOUR_CONNECTION_STRING"; // Replace with your actual connection string

// ...rest of your code...
```

Remember to replace placeholders with your actual username and password. For example:

```
const uri = `mongodb+srv://Yoshi:test123@...`
```

Important: Don't try connecting to my database; it will no longer exist by the time you watch this.

Save your changes and test the connection using Postman. You don't need to modify your Postman requests; your Node API is still running on `localhost`. The Node API will now communicate with MongoDB Atlas.

Initially, you won't see any books because the online database is empty. Let's add some using a POST request with your existing book data. You should see an `insertedId` and `acknowledged: true` in the response. Then, use a GET request to retrieve the books; you should see your newly added books! Verify this in Atlas by browsing collections; you should see a new collection called "Books" with your documents.

Conclusion and Future Plans

And that's it! We've successfully connected our database to MongoDB Atlas. This was surprisingly simple!

This concludes our MongoDB series. In the future, I'll create a MERN application playlist, showing you how to build a full-stack application with MongoDB, Node, Express, and React. I might even do tutorials on MongoDB Realm, a feature-rich platform for building mobile, web, and desktop applications, including authentication, serverless functions, a database, and real-time data synchronization. Stay tuned!

I really hope you enjoyed this series! If you did, please like, share, and subscribe. It means a lot!

For ad-free access to all my YouTube courses, premium courses, and early access, check out NetNinja Pro (netninja.dev). It's $9/month (half price for the first month with this promo code: `YOUR_PROMO_CODE`). You'll get ad-free access to every course, exclusive courses, access to my premium Udemy courses, and early access to all my YouTube courses. The signup link is below! See you in the next one!