Getting Started with MongoDB

This section helps you get started with MongoDB quickly. After completing the tutorial, you will have a basic understanding of MongoDB and how to use the mongo shell to interact with the MongoDB database server.

1

What is MongoDB (https://www.mongodbtutorial.org/getting-started/what-is-mongodb/)

Provide you with a quick introduction to MongDB and its main features

2

Install MongoDB (https://www.mongodbtutorial.org/getting-started/install-mongodb/)

Show you step by step how to install MongDB database server and Mongo Atlas tool/

3

MongoDB Basics (https://www.mongodbtutorial.org/getting-started/mongodb-basics/)

Explain to you some important concepts in MongoDB including documents, collections, databases, and namespaces.

4

MongoDB Shell (https://www.mongodbtutorial.org/getting-started/mongodb-shell/)

Introduce you to the MongoDB shell which is an interactive JavaScript interface to MongoDB that allows you to interact with MongoDB database server.

5

MongoDB Data Types (https://www.mongodbtutorial.org/getting-started/mongodb-data-types/)

Guide you on the most commonly used MongoDB data types.

What is MongoDB

Summary: in this tutorial, you will briefly learn about MongoDB and its features.

Introduction to MongoDB

MongoDB is an open-source, cross-platform, distributed document database. MongoDB is developed by MongoDB Inc (https://www.mongodb.com/company) . and categorized as a NoSQL database.

1) Easy to use

MongoDB is a document-oriented database. It uses the concept of the document to store data, which is more flexible than the row concept in the relational database management system (RDBMS).

A document allows you to represent complex hierarchical relationships with a single record.

MongoDB doesn't require predefined schemas that allow you to add to or remove fields from documents more quickly.

2) Designed to scale out

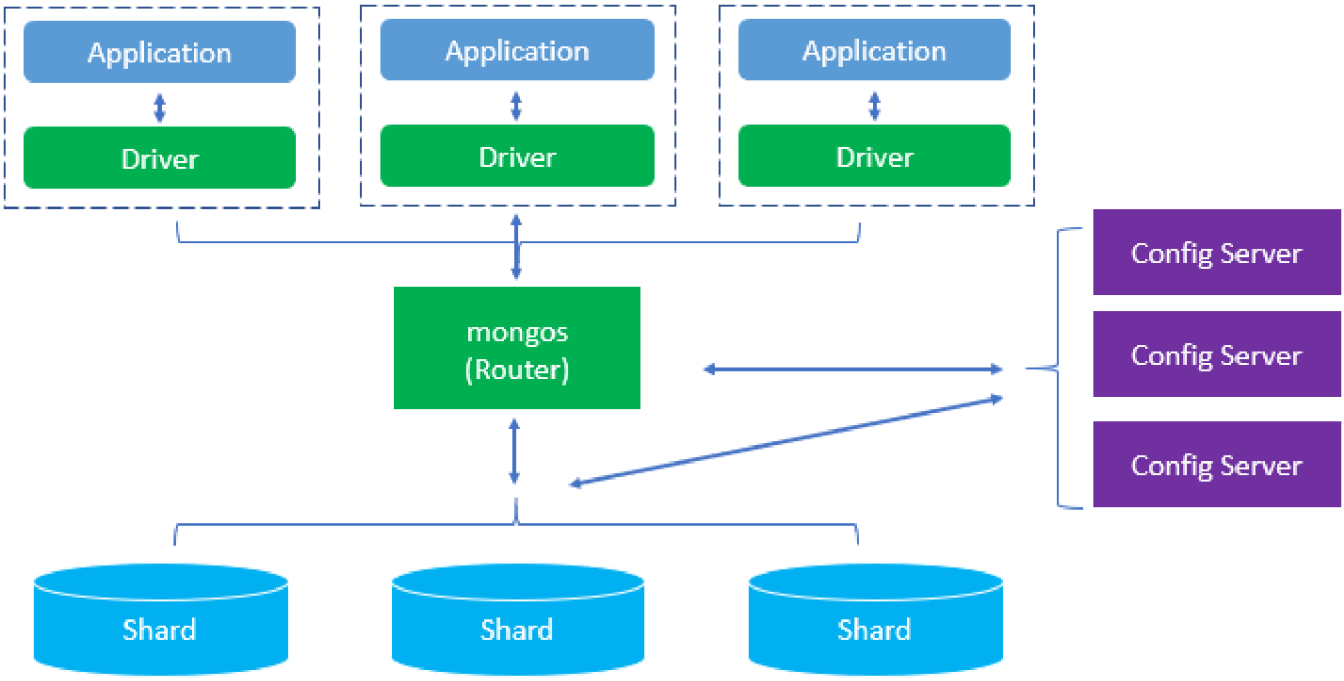
When the database grows, you'll have a challenge of how to scale it. There are two common ways:

- Scaling up – upgrade the current server to a bigger one with more resources (CPU, RAM, etc). However, getting a bigger server means increasing more costs.
- Scaling out – purchase additional servers and add them to the cluster. This is cheaper and more scalable than scaling up. The downside is that it takes more effort to manage multiple servers than a big one.

MongoDB was designed to scale out.

MongoDB allows you to split data across many servers. It also automatically manages the load balancing across the cluster, redistributing data, and routing updates to the correct servers.

The following picture illustrates how MongoDB scale-out using sharding across multiple servers:



3) Rich features

Like any database system, MongoDB allows you to insert, update, and delete, and select data. In addition, it supports other features including:

- Indexing
- Aggregation
- Specify collection and index types
- File Storage

Note that you will learn more about these features in detail in the next tutorial.

4) High performance

MongoDB was designed to maintain the high performance from both architecture and feature perspectives.

The philosophy of MongoDB is to create a full-featured database that is scalable, flexible, and fast.

MongoDB editions

MongoDB has three editions: community server, enterprise server, and atlas.

1) MongoDB Community Server

The MongoDB Community Edition is free and available on Windows, Linux, and macOS.

MongoDB Community Edition uses the Server Side Public License (https://www.mongodb.com/licensing/server-side-public-license) (SSPL). It means that if you offer MongoDB as a service to the public, you need to open the source code of the software that makes the service works e.g., administration and monitoring tools. Otherwise, you need to pay for the enterprise subscription.

If you use the MongoDB Community Edition as a component of your application, not the final product, you are free to use it.

2) MongoDB Enterprise Server

MongoDB Enterprise Server is a commercial edition of MongoDB as a part of the MongoDB Enterprise Advanced subscription.

3) MongoDB Atlas

MongoDB Atlas is a global cloud database service. It is a database as a service that allows you to focus on building apps rather than spending time managing the databases.

MongoDB Atlas is available on common cloud platforms such as AWS, Azure, and GCP. MongoDB Atlas has a free tier for your experiments.

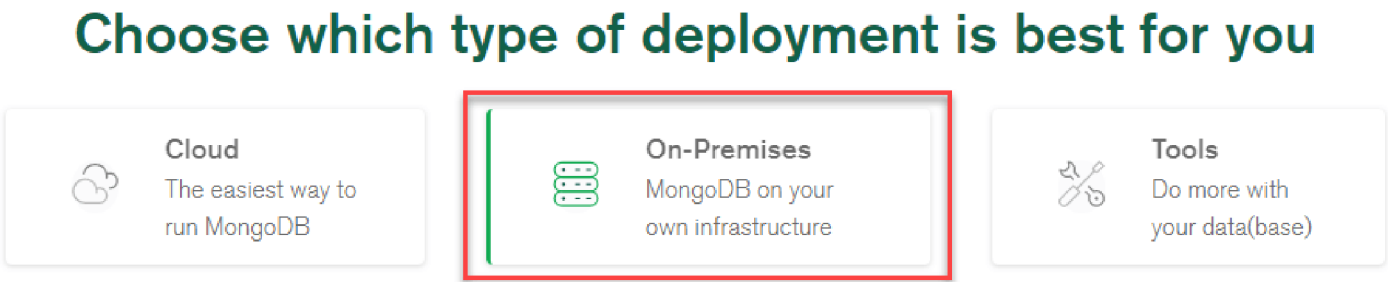
Install MongoDB

Summary: in this tutorial, you will learn how to install the MongoDB database server and its tools on your local system.

Download MongoDB Community Server

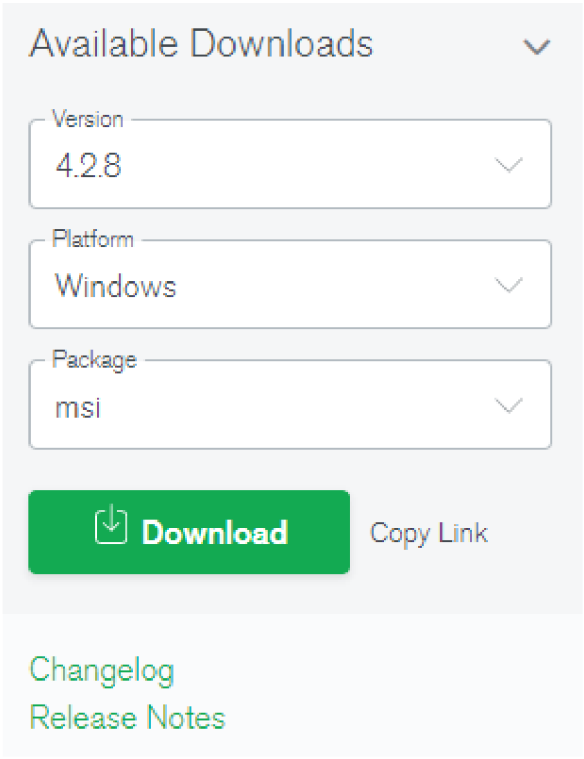
First, visiting the download page (https://www.mongodb.com/try/download/community) on the mongodb.com website.

Second, click the On-Premises (MongoDB on your own infrastructure tab)



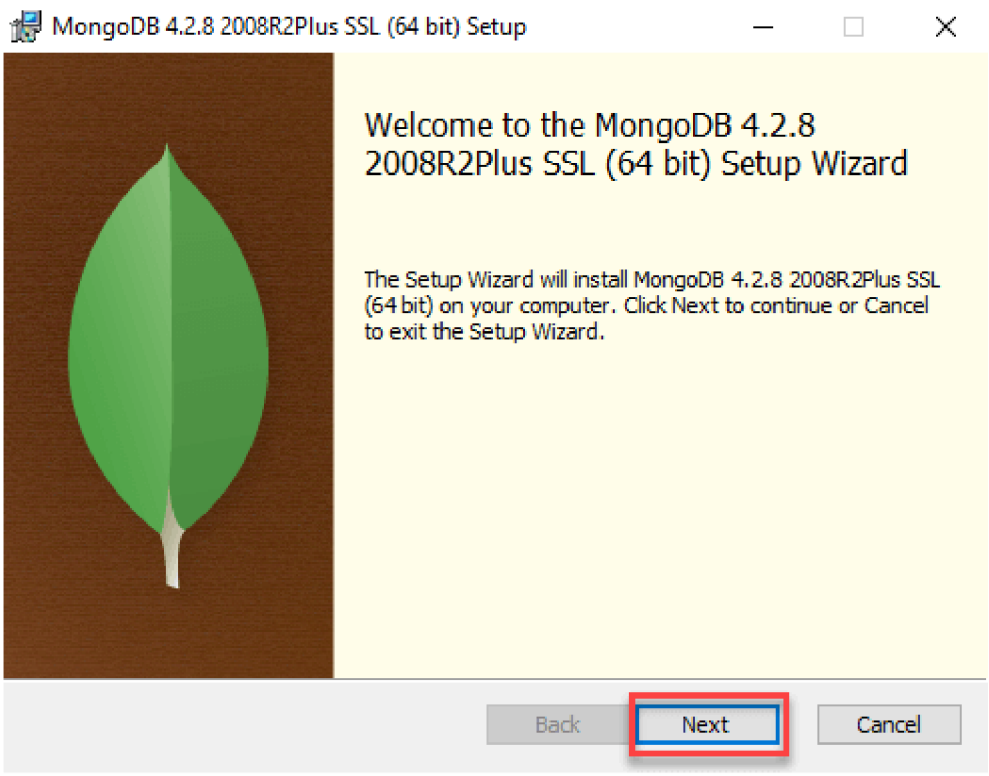
Third, select the MongDB Community Server.

Finally, select the version, platform, and click the download button to download the installation file.

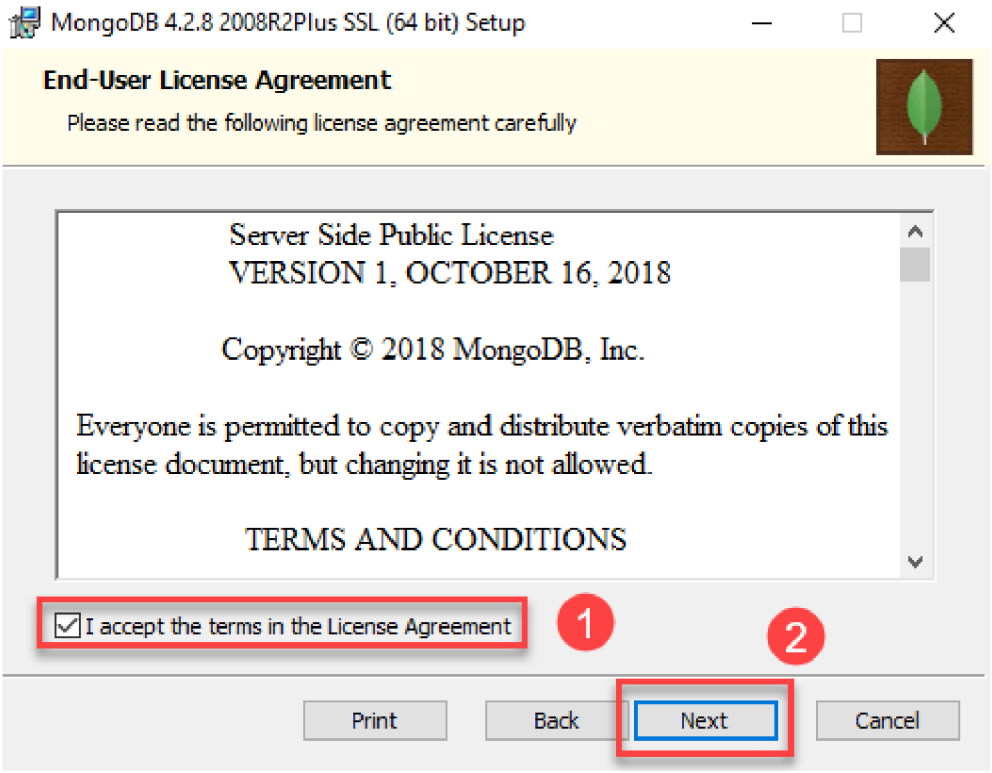


Install MongoDB Community Server on the local machine

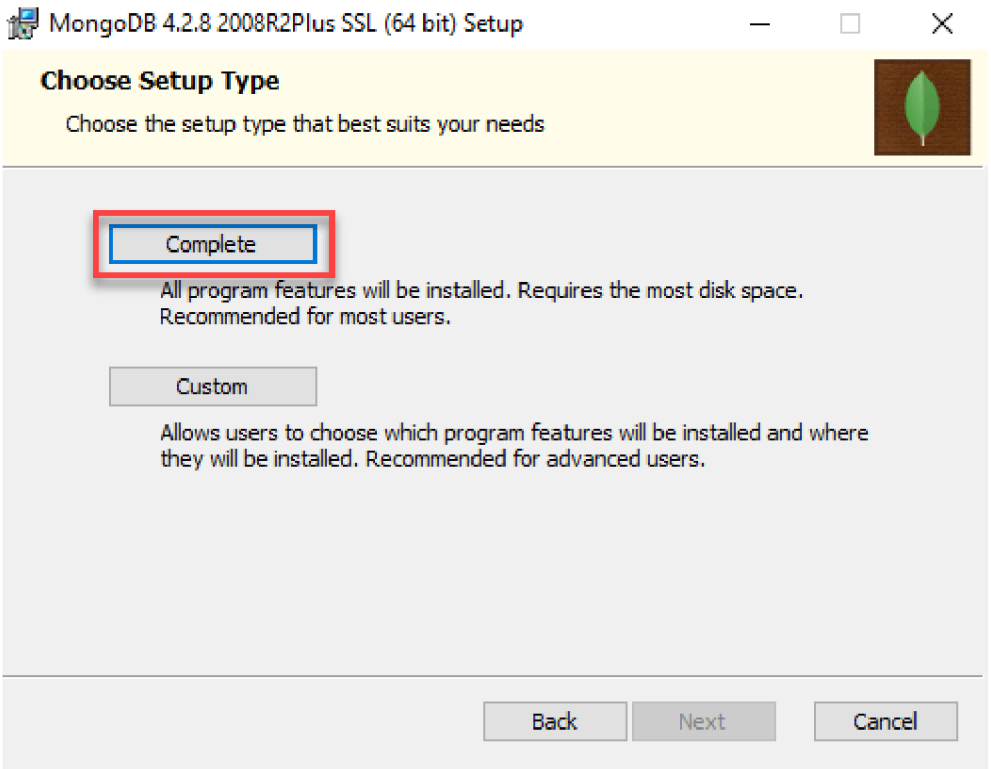
First, double-click the downloaded file to launch the setup wizard. Click the Next button to start setting up MongoDB.



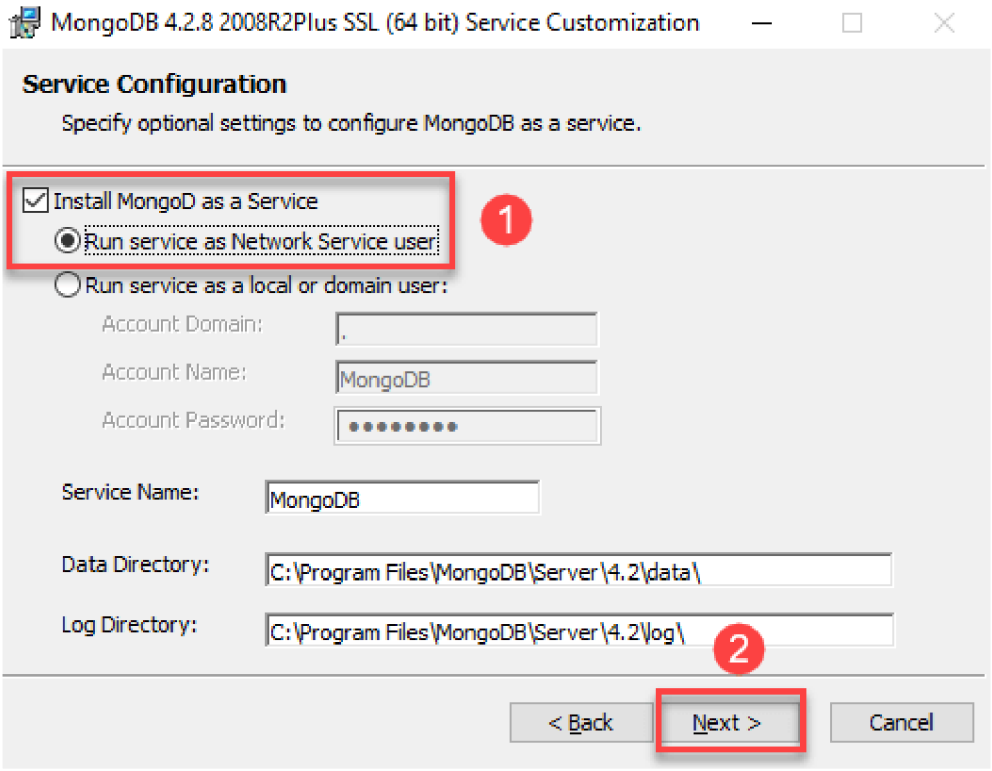
Second, accept the terms in the License Agreement and click the Next button:



Third, click the **Complete** button to install all program features. If you want to select which features to install and where they will be installed, select the Custom button. However, this option is recommended for advanced users only.



Fourth, select Install MongoD as a service and click the Next button.

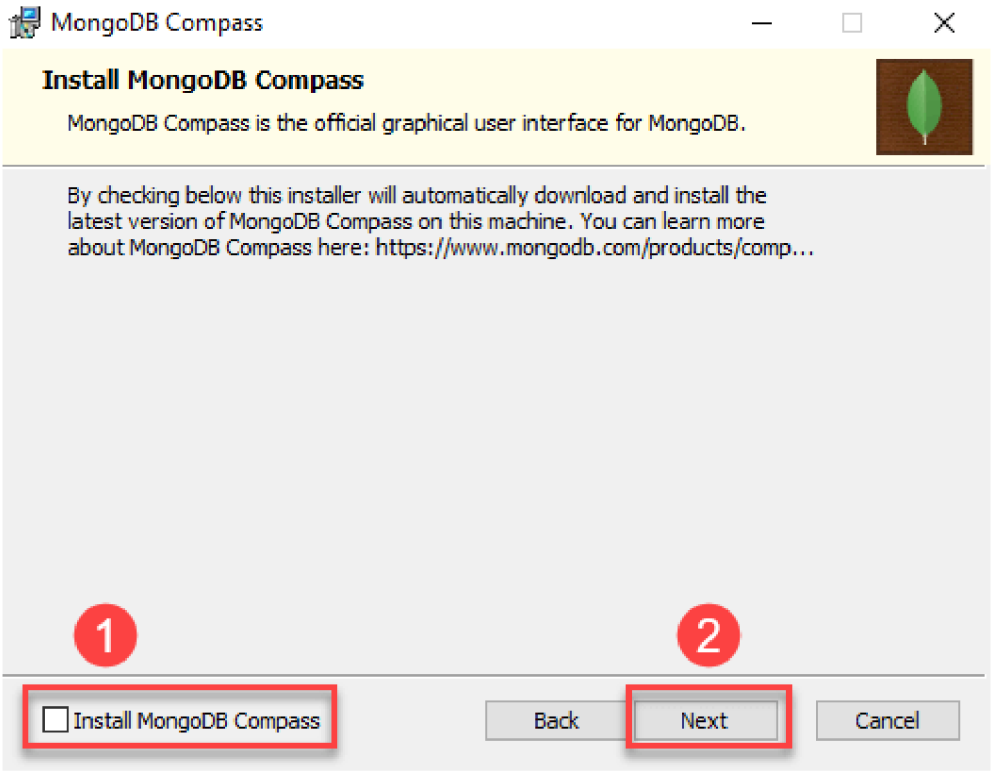


Fifth, uncheck the **Install MongoDB Compass** checkbox and click the **Next** button. If you select it, the setup wizard will also install the MongoDB Compass with the community version.

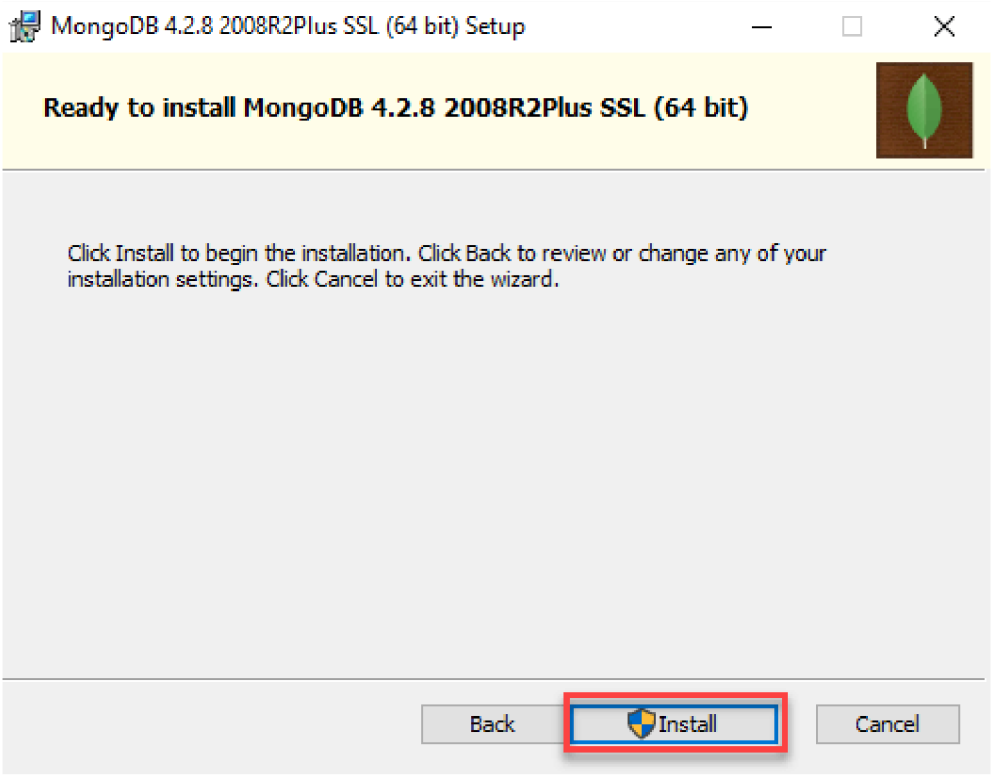
> MongoDB Compass is a GUI tool that allows you to interact with MongoDB server including querying, indexing, document validation, and more.

> MongoDB Compass has several versions. Like MongoDB, the community version always is free, but with limited features.

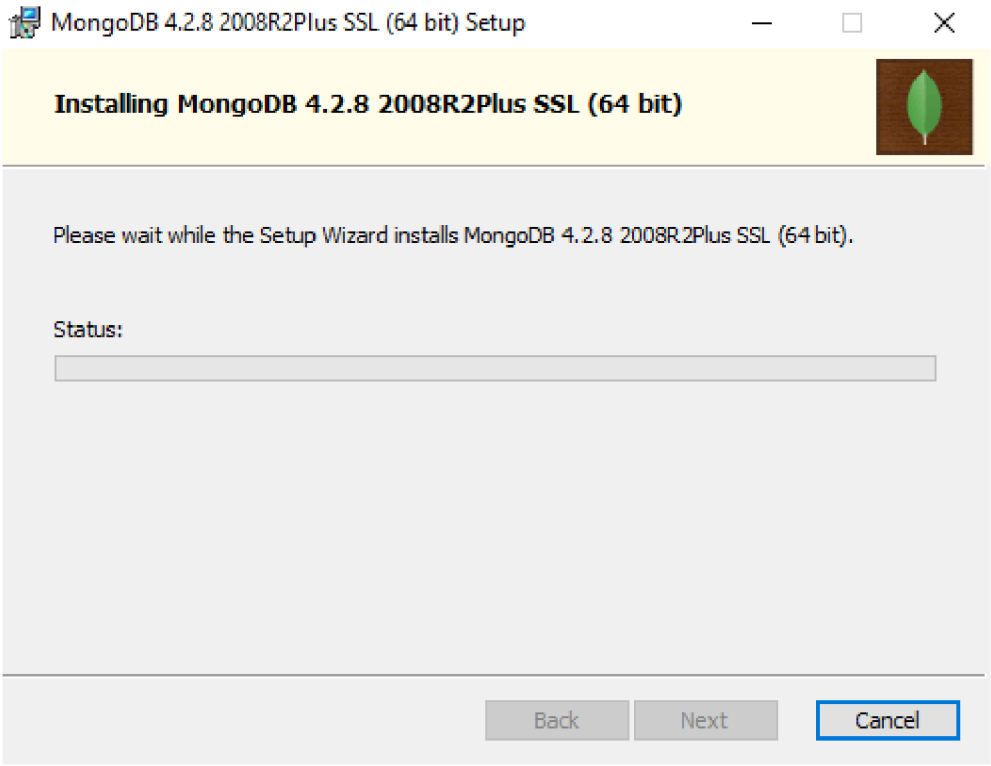
> The good news is that starting from April 30, 2020, the full version of MongoDB Compass is free for all (https://www.mongodb.com/blog/post/compass-now-free-for-all) . You will learn how to download and install MongoDB Compass in the next section.



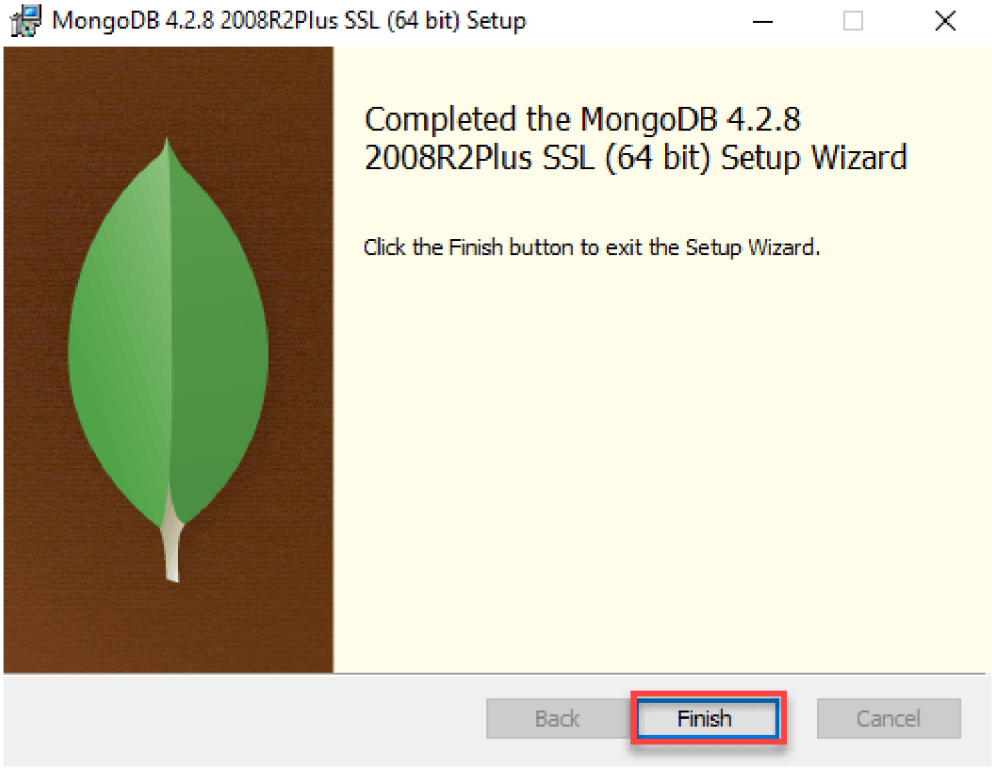
Sixth, click the Install button to start the installation.



The install will take some minutes to complete.



Seventh, click the **Finish** button to exit the setup wizard.



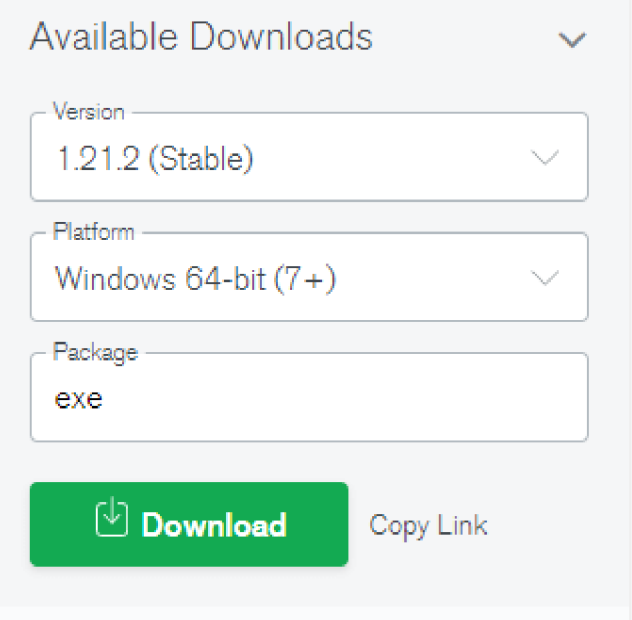
Download MongoDB Compass

You follow these steps to download MongoDB Compass:

First, visit the download page on the mongodb.com website (https://www.mongodb.com/try/download/compass) .

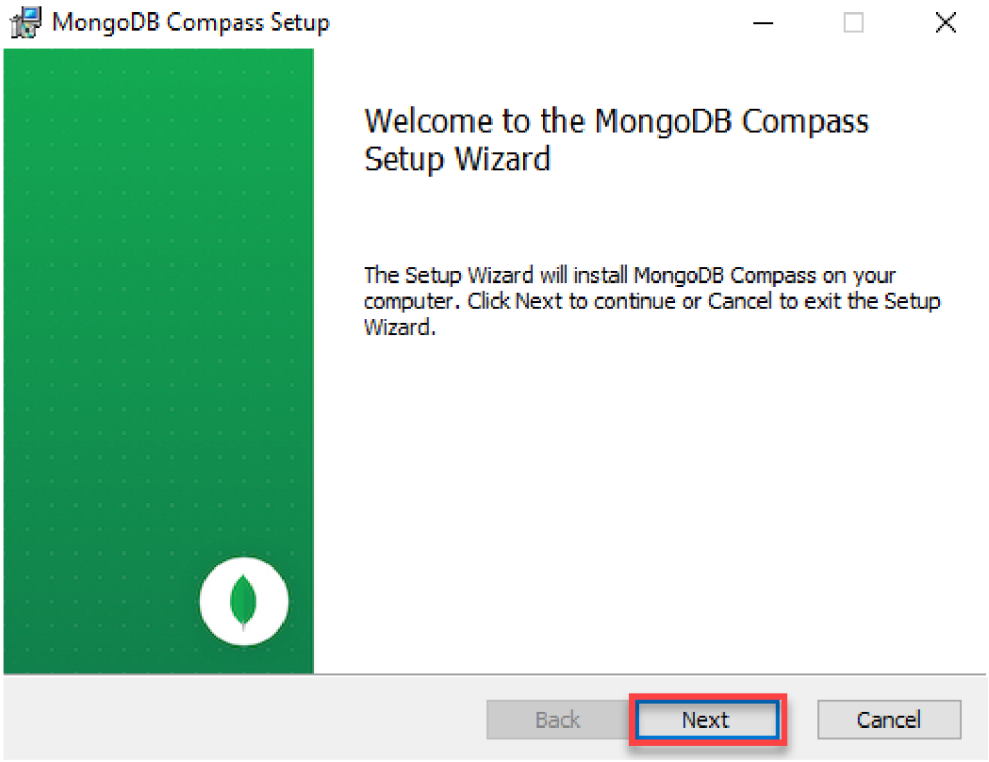
Second, select the MongoDB Compass tool.

Third, select the stable version to download. Note that the version that you see on the website may be higher than the version on the following screenshot:



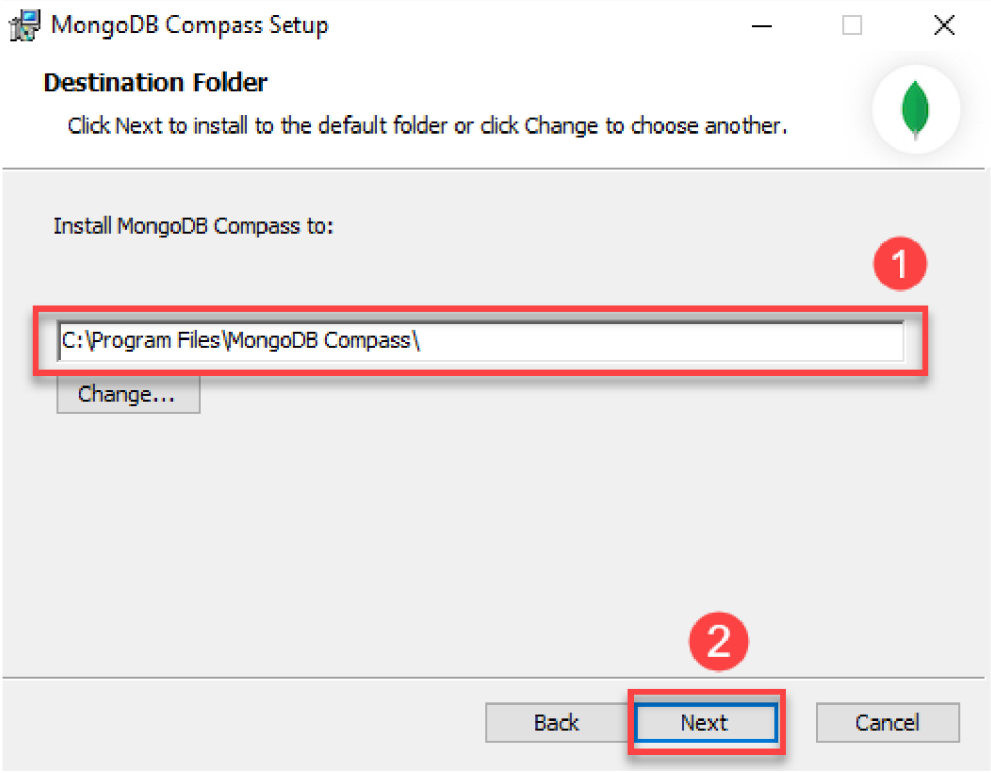
Install MongoDB Compass

First, double-click the installer file to launch the MongoDB Compass Setup Wizard and click the Next button.

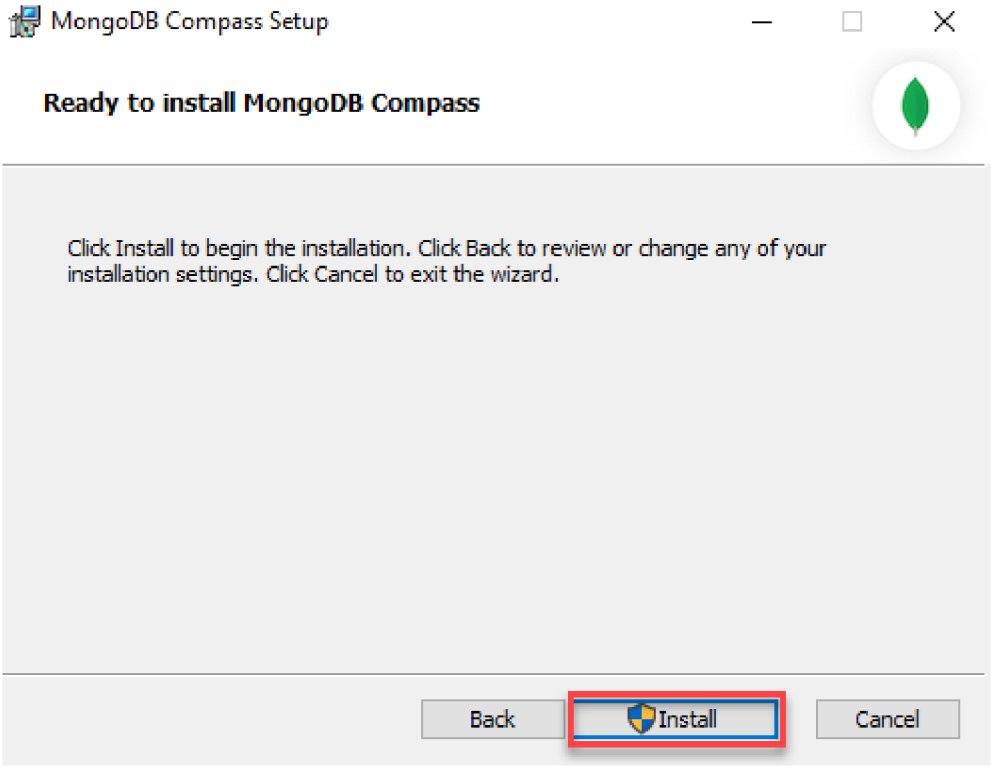


Second, select the destination folder and click Next to install to the default folder which is

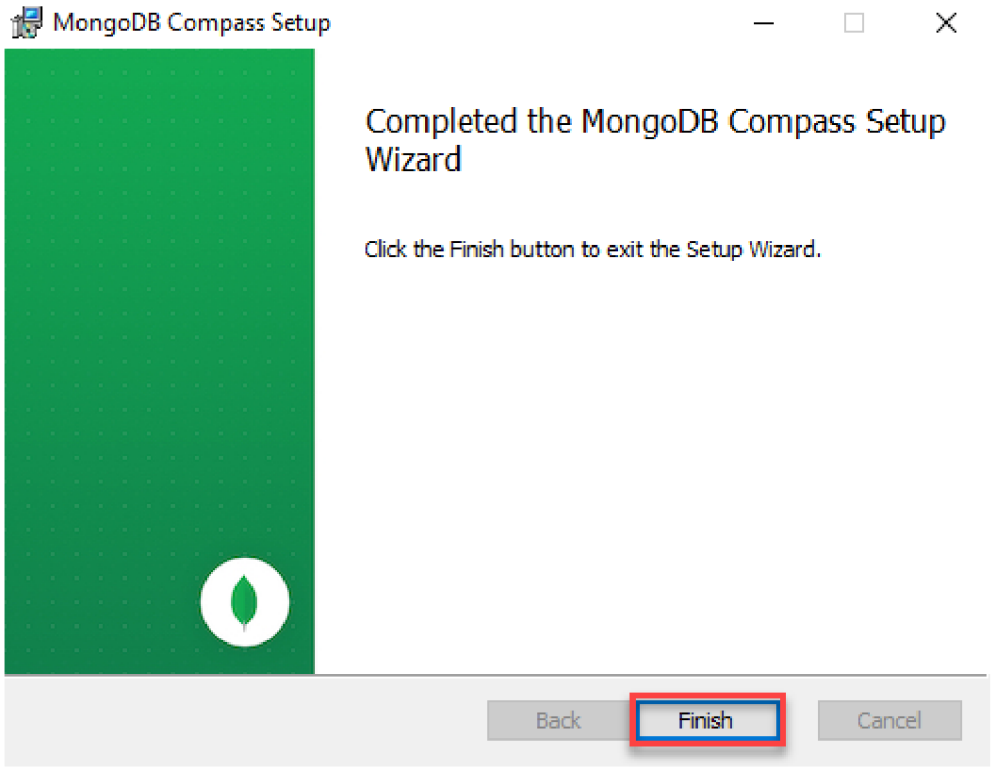
```
C:\Program Files\MongoDB Compass
```



Third, click the Install button to begin the installation.



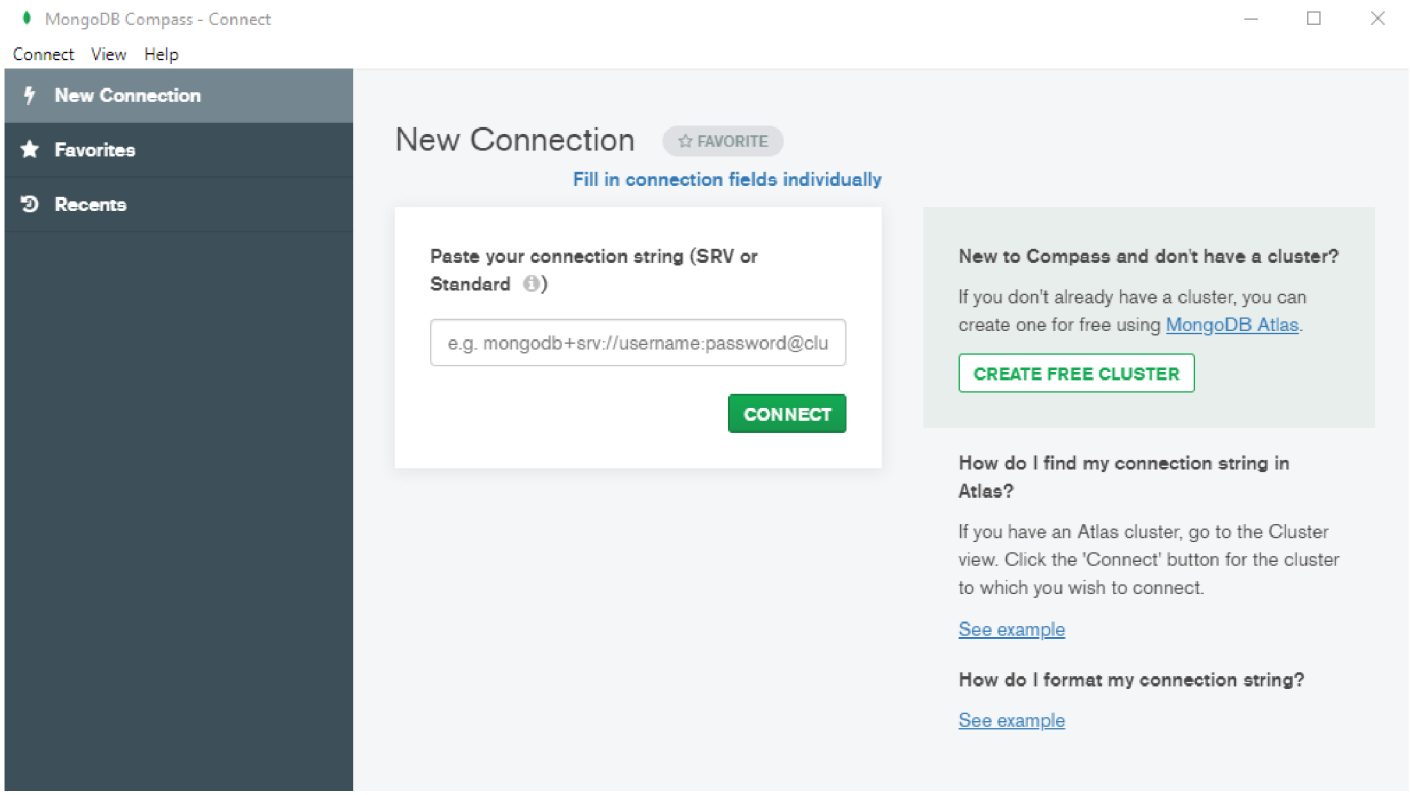
Fourth, once the installation is completed, click the Finish button to exit the Setup Wizard.



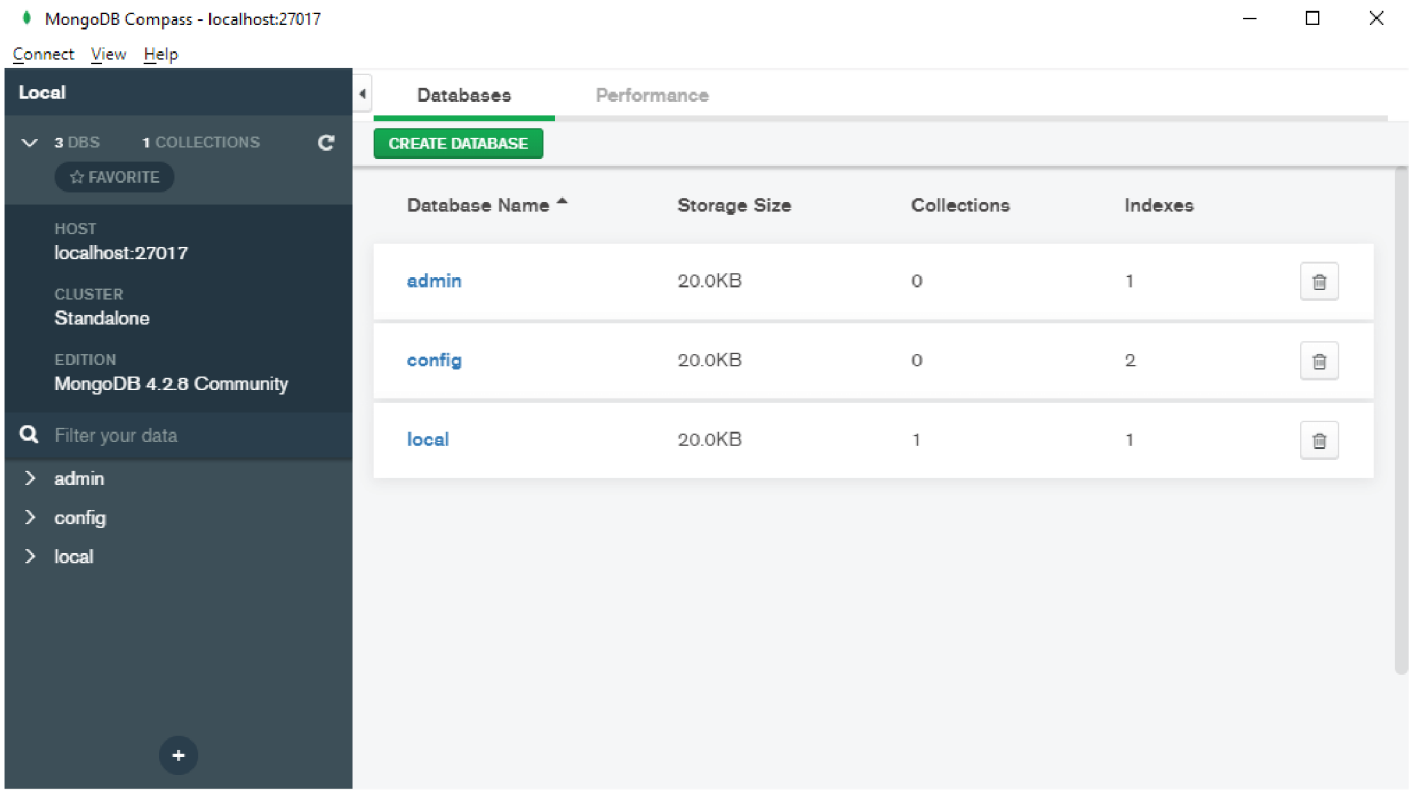
You'll find the MongoDB Compass in the Start menu.

To launch the MongoDB Compass, click the MongoDB Compass icon. The following shows the main screen of the MongoDB Compass.

To connect to the local MongoDB Server, click the **Connect** button:



If everything is fine, you will see the following screen:



In this tutorial, you've learned step by step how to install the MongoDB server and MongoDB Compass on your computer. Now, you're ready to learn about MongoDB.

MongoDB Basics

Summary: in this tutorial, you'll learn some basic concepts of MongoDB such as documents, collections, databases, and namespaces.

Data formats

In MongoDB, you will often deal with JSON and BSON formats. Therefore, it's important to fully understand them.

JSON

JSON stands for JavaScript Object Notation. JSON syntax is based on a subset of JavaScript ECMA-262 3rd edition.

A JSON document is a collection of fields and values in a structured format. For example:

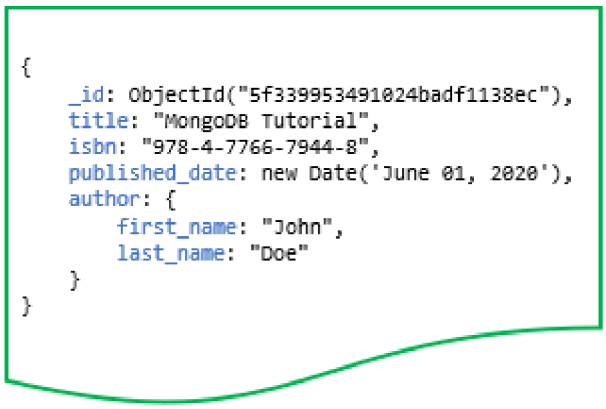
```
{
   "first_name": "John",
   "last_name": "Doe",
   "age": 22,
   "skills": ["Programming","Databases", "API"]
}
```

BSON

BSON stands for Binary JSON, which is a binary-coded serialization of JSON-like documents.

Documents

MongoDB stores data records as BSON documents, which are simply called documents.



A document is a set of field-and-value pairs with the following structure:

```
{
   field_name1: value1,
   field_name2: value2,
   field_name3: value3,
   ...
}
```

In this syntax, the field names are strings and values can be numbers, strings, objects, arrays, etc. For example:

```
{
    _id: ObjectId("5f339953491024badf1138ec"),
    title: "MongoDB Tutorial",
    isbn: "978-4-7766-7944-8",
    published_date: new Date('June 01, 2020'),
    author: { first_name: "John"
, last_name: "Doe"}
}
```

This document has the following field-and-value pairs:

- `_id` holds an `ObjectId`
- `title` holds a string.
- `isbn` holds a string.
- `published_date` holds a value of the `Date` type.

- `author` holds an embedded document that contains two fields `first_name` and `last_name` .

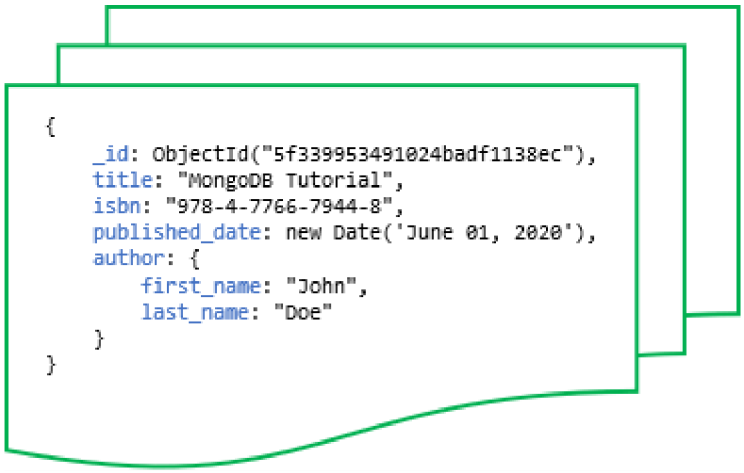
If you are familiar with a relational database management system (RDBMS), you will find that a document is similar to a row in a table, but it is much more expressive.

Field names have the following restrictions:

- MongoDB reserves the field `_id` and uses it to uniquely identify the document.
- Field names cannot contain null characters.
- Top-level field names cannot start with the dollar sign (`$`) character.

Collections

MongoDB stores documents in a collection. A collection is a group of documents.



A collection is analogous to a table in an RDBMS.

MongoDB	**RDBMS**
Documents	Rows
Collections	Tables

Unlike a table that has a fixed schema, a collection has a dynamic schema.

It means that a collection may contain documents that have any number of different “shapes”. For example, you can store the following documents in the same collection:

```
{
    title: "MongoDB Tutorial",
    published_date: new Date('June 01, 2020')
}

{
    title: "MongoDB Basics",
    published_date: new Date('Jan 01, 2021'),
    isbn": "978-4-7766-7944-8"
}
```

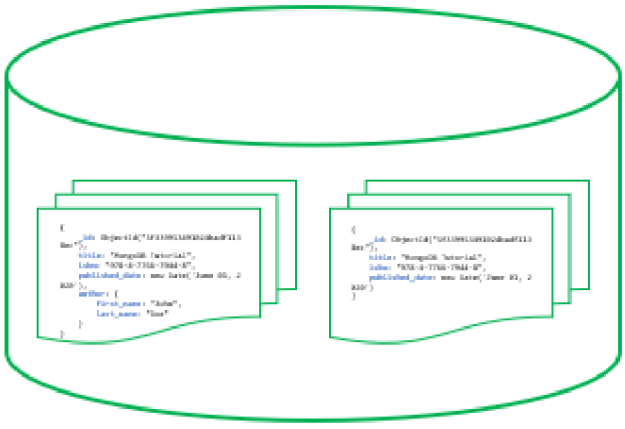
Note that the second document has one more field than the first one. In theory, you can have completely different fields for every document.

A collection has a name e.g., `books` . The collection name cannot:

- contain the dollar sign (`$`)
- contain the null character (`\0`).
- be an empty string.
- begin with the system because MongoDB reserves system* for internal collection names.

Databases

MongoDB stores collections into a database. A single instance of MongoDB can host multiple databases.



A database can be referenced by a name for example `bookdb` . The database names cannot:

- Be an empty string (`""`).
- Contain any of these characters: */*, \, ., ", *, <, >, :, |, *?*, *$*, (a single space), or *\0* (the `null` character).
- Exceed the maximum size which is 64 bytes.

MongoDB also has some reserved database names such as `admin` , `local` , and `config` that you cannot use to create new databases.

Namespace

A namespace is a concatenation of the database name with a collection in that database. Namespaces allow you to fully qualify collections.

For example, if the collection name is `books` and database name is `bookdb` , the namespace of the `books` collection would be `bookdb.books` .

Summary

- MongoDB stores data records as BSON documents. A document is a set of field-and-value pairs.
- MongoDB stores documents in a collection and collections in a database.
- A namespace is a concatenation of the database name and the collection name (`database_name.collection_name`) to fully qualify the collection.

MongoDB Shell

Summary: in this tutorial, you will learn about the mongo shell and how to use it to interact with the MongoDB database server.

Introduction to the mongo shell

The mongo shell is an interactive JavaScript interface to MongoDB. The mongo shell allows you to manage data in MongoDB as well as carry out administrative tasks.

> The mongo shell is similar to the mysql in MySQL, psql in PostgreSQL, and SQL*Plus in Oracle Database.

> Note that the `mongo` shell that is included in the MongoDB installation by default was deprecated in MongoDB v5.0. The `mongsh` is the replacement of the legacy `mongo` shell.

Before using the mongo shell, you need to download and install it (https://docs.mongodb.com/mongodb-shell/) .

To start the mongo shell, you open the Terminal on macOS and Linux or a Command Prompt on Windows and use the following command:

```
mongosh
```

> The mongo shell automatically connects to the MongoDB running on the local server (`localhost`) with the default port (`27017`). Therefore, you need to ensure that the MongoDB server is up and listening on this port before starting the mongo shell.

Note that you can use the mongo shell as a full-featured JavaScript interpreter and as a MongoDB client.

JavaScript interpreter

The mongo shell is a full-featured JavaScript interpreter so that you can execute any JavaScript code like this:

```
> Math.max(10,20,30);
30
```

The `mongo` shell allows you to enter multiline commands. It will detect if the JavaScript statement is complete when you press `enter` .

If the statement is not complete, the mongo shell will allow you to type on the next line after the three dots (`...`).

```
> function add(a, b) {
... return a + b;
... }
> add(10,20);
30
```

To clear the screen, you use the console.clear() like this:

```
console.clear()
```

MongoDB client

The mongo shell is a MongoDB client. By default, the mongo shell connects to the `test` database on the MongoDB server and assigns the database connection to the global variable called `db` .

The `db` variable allows you to see the current database:

```
> db
test
```

> Note that when you type a variable or an expression on the command line, the mongo shell will evaluate it and returns the result.

Besides supporting JavaScript syntax, the `mongosh` shell provides you with useful commands that easily interact with the MongoDB database server.

For example, you can use the `shows dbs` command to list all the databases on the server:

```
test> show dbs
admin        41 kB
config     73.7 kB
local      81.9 kB
```

The output shows three databases on the server.

To switch the current database to another, you use the `use <database>` command. For example, the following command switches the current database to the `bookdb` database:

```
test> use bookdb
switched to db bookdb
```

Note that you can switch to a database that does not exist. In this case, MongoDB will automatically create that database when you first save the data in it.

The mongo shell now assigns `bookdb` to the `db` variable:

```
> db
bookdb
```

Now, you can access the `books` collection from the `bookstore` database via the `db` variable like this:

```
> db.books
bookstore.books
```

Basic CRUD operations

The following section shows you how to create (C), read (R), update (U), and delete (D) a document. These operations are often referred to as CRUD operations.

> Note that this section only covers the basic CRUD operations (https://www.mongodbtutorial.org/mongodb-crud/) , you will learn about them in detail in the CRUD tutorial (https://www.mongodbtutorial.org/mongodb-crud/) .

Create

To add a document to a collection, you use the `insertOne()` method of the collection.

The following command adds a new document (or a new book) to the `books` collection:

```
db.books.insertOne({
   title: "MongoDB Tutorial",
   published_year: 2020
})
```

Output:

```
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5f2f39fb82f5c7bd6c9375a8")
}
```

Once you press `enter` , the mongo shell sends the command to the MongoDB server.

If the command is valid, MongoDB inserts the document and returns the result. In this case, it returns an object that has two keys `acknowledged` and `insertedId` .

The value of the `insertedId` is the value of the `_id` field in the document.

When you add a document to a collection without specifying the `_id` field, MongoDB automatically assigns a unique `ObjectId` value to the `_id` field and add it to the document.

MongoDB uses the `_id` field to uniquely identify the document in the collection.

Read

To select documents in a collection, you use the `findOne()` method:

```
db.books.findOne()
```

Output:

```
{
  _id: ObjectId("62143f34cca1c7af0ad1d126"),
  title: 'MongoDB Tutorial',
  published_year: 2020
}
```

To format the output, you use the `pretty()` method like this:

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

Output:

```
{
        "_id" : ObjectId("5f2f3d8882f5c7bd6c9375ab"),
        "title" : "MongoDB Tutorial",
        "published_year" : 2020
}
```

As you can see clearly from the output, MongoDB added the `_id` field together with other field-and-value pairs to the document.

Update

To update a single document, you use the `updateOne()` method.

The `updateOne()` method takes at least two arguments:

- The first argument identifies the document to update.
- The second argument represents the updates that you want to make.

The following shows how to update the `published_year` of the document whose title is `"MongoDB Tutorial"`:

```
db.books.updateOne(
        { title: "MongoDB Tutorial"},
        { $set: { published_year: 2019 }}
)
```

Output:

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

How it works.

The first argument identifies which document to update. In this case, it will update the first document that has the title `"MongoDB tutorial"`:

```
{title: "MongoDB Tutorial"}
```

The second argument is an object that specifies which fields in the document to update:

```
{
    $set: {
        published_year: 2019
    }
}
```

The `$set` is an operator that replaces a field value with a specified value. In this example, it updates the `published_year` of the document to `2019` .

Delete

To delete a document from a collection, you use the `deleteOne()` method. The `deleteOne()` method takes one argument that identifies the document that you want to delete.

The following example uses the `deleteOne()` method to delete a document in the `books` collection:

```
db.books.deleteOne({title: "MongoDB Tutorial"});
```

Output:

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

The output shows that one document has been deleted successfully via the `deletedCount` field.

To show all collections of the current database, you use the show collections command:

```
show collections
```

Output:

```
books
```

Currently, the `bookdb` database has one collection which is `books` .

Summary

- The mongo shell is an interactive JavaScript interface to MongoDB.

- The mongo shell allows you to manage data in MongoDB as well as perform administrative tasks.
- Use the `show dbs` command to list all databases on the server.
- Use the `use` command to switch to another database.
- Use the `show collections` to list all collections of the current database.
- CRUD is referred to as four basic functions including creating, reading, updating, and deleting data.

MongoDB Data Types

Summary: in this tutorial, you will learn about the most commonly used MongoDB data types.

Null

The `null` type is used to represent a `null` and a field that does not exist. For example:

```
{
   "isbn": null
}
```

Boolean

The boolean type has two values `true` and `false` . For example:

```
{
   "best_seller": true
}
```

Number

By default, the mongo shell uses the 64-bit floating-point numbers. For example:

```
{
    "price": 9.95,
    "pages": 851
}
```

The `NumberInt` and `NumberLong` classes represent 4-byte and 8-byte integers respectively. For example:

```
{
   "year": NumberInt("2020"),
   "words": NumberLong("95403")
}
```

String

The string type represents any string of UTF-8 characters. For example:

```
{
   "title": "MongDB Data Types"
}
```

Date

The date type stores dates as 64-bit integers that represents milliseconds since the Unix epoch (January 1, 1970). It does not store the time zone. For example:

```
{
    "updated_at":  new Date()
}
```

In JavaScript, the `Date` class is used to represent the date type in MongoDB.

Note that you should always call the `new Date()` , not just `Date()` when you create a new `Date` object because the `Date()` returns a string representation of the date, not the `Date` object.

The mongo shell displays dates using local time zone settings. However, MongoDB does not store date with the time zone. To store the time zone, you can use another key e.g., `timezone` .

Regular Expression

MongoDB allows you to store JavaScript regular expressions (https://www.javascripttutorial.net/javascript-regex/) . For example:

```
{
    "pattern": /\d+/
}
```

In this example, `/\d+/` is a regular expression that matches one or more digits.

Array

The array type allows you to store a list of values of any type. The values do not have to be in the same type, for example:

```
{
    "title": "MongoDB Array",
    "reviews": ["John", 3.5, "Jane", 5]
}
```

The good thing about arrays in the document is that MongoDB understands their structures and allows you to carry operations on their elements.

For example, you can query all documents where `5` is an element of the `reviews` array. Also, you can create an index on the `reviews` array to improve the query performance.

Embeded Document

A value of a document can be another document that is often referred to as an embedded document.

The following example shows a `book` document that contains the `author` document as an embedded document:

```
{
    "title": "MongoDB Tutorial",
    "pages": 945,
    "author": {
       "first_name": "John",
       "last_name": "Doe"
```

```
    }
}
```

In this example, the `author` document has its own key/value pairs including `first_name` and `last_name`.

Object ID

In MongoDB, every document has an `"_id"` key. The value of the `"_id"` key can be any type. However, it defaults to an `ObjectId`.

The value of the `"_id"` key must be unique within a collection so that MongoDB can identify every document in the collection.

The `ObjectId` class is the default type for `"_id"`. It is used to generate unique values globally across servers. Since MongoDB is designed to be distributed, it is important to ensure the identifiers are unique in the shared environment.

The `ObjectId` uses 12 bytes for storage, where each byte represents 2 hexadecimal digits. In total, an `ObjectId` is 24 hexadecimal digits.

The 12-byte ObjectId value consists of:

- A 4-byte timestamp value that represents the `ObjectId`'s generated time measured in seconds since the Unix epoch.
- A 5-byte random value.
- A 3-byte increment counter, initialized to a random value.

These first 9 bytes of an `ObjectId` guarantee its uniqueness across servers and processes for a single second. The last 3 bytes guarantee uniqueness within a second in a single process.

As a result, these 12-bytes allow for up to 256^3 (16,777,216) unique `ObjectId` s values to be generated per process in a single second.

When you insert a document without specifying a value for the `"_id"` key, MongoDB automatically generates a unique id for the document. For example:

```
db.books.insertOne({
   "title": "MongoDB Basics"
```

```
});
```

Output:

```
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5f2fcae09b58c38603442a4f")
}
```

MongoDB generated the id with the value `ObjectId("5f2fcae09b58c38603442a4f")` . You can view the inserted document like this:

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

Output:

```
{
    "_id" : ObjectId("5f2fcae09b58c38603442a4f"),
     "title" : "MongoDB Basics"
}
```

In this tutorial, you have learned the most commonly used MongoDB data types including null, number, string, array, regular expression, date, and ObjectId.

mongoDB TUTORIAL

MongoDB CRUD

This section covers the basics of CRUD operations, including:

- Adding new documents to a collection
- Selecting documents
- Updating existing documents
- Removing documents from a collection

Section 1. Adding new documents to a collection

- insertOne() (https://www.mongodbtutorial.org/mongodb-crud/mongodb-insertone/) – show you how insert a single document into a collection.
- insertMany() (https://www.mongodbtutorial.org/mongodb-crud/mongodb-insertmany/) – learn how to insert multiple documents into a collection.

Section 2. Selecting documents

- findOne() (https://www.mongodbtutorial.org/mongodb-crud/mongodb-findone/) – show you how to query a single document.
- find() (https://www.mongodbtutorial.org/mongodb-crud/mongodb-find/) – learn how to query documents from a collection.
- Projection (https://www.mongodbtutorial.org/mongodb-crud/mongodb-projection/) – guide you on how to specify the fields to return in the matching document.

Section 3. Selecting documents using comparison query operators

- $eq (https://www.mongodbtutorial.org/mongodb-crud/mongodb-eq/) – select documents where the value of a field is equal to a specified value.
- $gt (https://www.mongodbtutorial.org/mongodb-crud/mongodb-gt/) – select documents where the value of a field is greater than a specified value.
- $gte (https://www.mongodbtutorial.org/mongodb-crud/mongodb-gte/) – select documents where the value of a field is greater than or equal to a specified value.
- $lt (https://www.mongodbtutorial.org/mongodb-crud/mongodb-lt/) – select documents where the value of a field is less than a specified value.
- $lte (https://www.mongodbtutorial.org/mongodb-crud/mongodb-lte/) – select documents where the value of a field is less than or equal to a specified value.
- $ne (https://www.mongodbtutorial.org/mongodb-crud/mongodb-ne/) – select documents where the value of a field is not equal to a specified value.
- $in (https://www.mongodbtutorial.org/mongodb-crud/mongodb-in/) – select documents where the value of a field equals any value in an array
- $nin (https://www.mongodbtutorial.org/mongodb-crud/mongodb-nin-not-in/) – select documents where the value of a field doesn't equal any value in an array

Section 4. Selecting documents using logical query operators

- $and (https://www.mongodbtutorial.org/mongodb-crud/mongodb-and/) – show you how to perform a logical AND operator to select documents.
- $or (https://www.mongodbtutorial.org/mongodb-crud/mongodb-or/) – guide you on how to perform a logical OR operator to find documents.

- $not (https://www.mongodbtutorial.org/mongodb-crud/mongodb-not/) – learn how to perform a logical NOT operator to query documents.
- $nor (https://www.mongodbtutorial.org/mongodb-crud/mongodb-nor/) – show you how to use the logical NOR operator to select documents.

Section 5. Selecting documents using element query operators

- $exists (https://www.mongodbtutorial.org/mongodb-crud/mongodb-exists/) – select documents where a field exists.
- $type (https://www.mongodbtutorial.org/mongodb-crud/mongodb-type/) – select documents where the value of a field is an instance of a BSON type.

Section 6. Querying arrays

- $size (https://www.mongodbtutorial.org/mongodb-crud/mongodb-size/) – select documents if they have an array field with a specified size.
- $all (https://www.mongodbtutorial.org/mongodb-crud/mongodb-all/) – select documents that have an array with all elements specified in a query.
- $elemMatch (https://www.mongodbtutorial.org/mongodb-crud/mongodb-elemmatch/) – select documents if the element in the array field matches all the specified queries.

Section 7. Sorting & Limiting

- sort() (https://www.mongodbtutorial.org/mongodb-crud/mongodb-sort/) – show you how to sort returned documents by one or more fields in ascending and descending orders.

- limit() (https://www.mongodbtutorial.org/mongodb-crud/mongodb-limit/) – learn how to specify the number of documents returned by a query.

Section 8. Updating documents

- updateOne() (https://www.mongodbtutorial.org/mongodb-crud/mongodb-updateone/) – show you how to update a single document that satisfies a condition.
- updateMany() (https://www.mongodbtutorial.org/mongodb-crud/mongodb-updatemany/) – learn how to update multiple documents that match a condition.
- $inc (https://www.mongodbtutorial.org/mongodb-crud/mongodb-update-inc/) – increase/decrease field values
- $min (https://www.mongodbtutorial.org/mongodb-crud/mongodb-min/) – update the value of a field to a specified value if the specified value is less than the current value of the field.
- $max (https://www.mongodbtutorial.org/mongodb-crud/mongodb-max/) – update the value of a field to a specified value if the specified value is greater than the current value of the field.
- $mul (http://$mul) – multiply the value of a field by a number.
- $unset (https://www.mongodbtutorial.org/mongodb-crud/mongodb-remove-field-unset/) – remove one or more fields from a document.
- $rename (https://www.mongodbtutorial.org/mongodb-crud/mongodb-rename-field/) – rename a field in a document

Section 9. Deleting documents

- deleteOne() (https://www.mongodbtutorial.org/mongodb-crud/mongodb-deleteone/) – delete a single document from a collection.
- deleteMany() (https://www.mongodbtutorial.org/mongodb-crud/mongodb-deletemany/) – delete all documents that match a condition from a collection.

MongoDB insertOne

Summary: in this tutorial, you will learn how to use the MongoDB `insertOne()` method to insert a single document into a collection.

Introduction to MongoDB insertOne() method

The `insertOne()` method allows you to insert a single document into a collection.

The `insertOne()` method has the following syntax:

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

The `insertOne()` method accepts two arguments:

- `document` is a document that you want to insert into the `collection`. The `document` argument is required.
- `writeConcern` is an optional argument that describes the level of acknowledgment requested from MongoDB for insert operation to a standalone MongoDB server or to shared clusters. We'll discuss the writeConcern another tutorial.

The `insertOne()` method returns a document that contains the following fields:

- `acknowledged` is a boolean value. It is set to true if the insert executed with write concern or false if the write concern was disabled.
- `insertedId` stores the value of `_id` field of the inserted document.

Note that if the `collection` does not exist, the `insertOne()` method will also create the collection and insert the `document` into it.

If you don't specify the `_id` field in the document, MongoDB will add the `_id` field and generate a unique `ObjectId` for it before insert.

If you explicitly specify a value for the `_id` field, you need to ensure that it is unique in the collection. Otherwise, you will get a duplicate key error.

To insert multiple documents into a collection, you use the `insertMany()` (https://www.mongodbtutorial.org/mongodb-crud/mongodb-insertmany/) method.

MongoDB insertOne() method examples

First, you need to launch the mongo shell and connect it to the `bookdb` database:

```
mongosh bookdb
```

1) Insert a document without an _id field example

The following example uses the `insertOne()` method to insert a new document into the books collection:

```
db.books.insertOne({
    title: 'MongoDB insertOne',
    isbn: '0-7617-6154-3'
});
```

Output:

```
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5f31cf00902f22de3464ddc4")
}
```

In this example, we passed a document to the `insertOne()` method without specifying the `_id` field. Therefore, MongoDB automatically added the `_id` field and assigned it a unique `ObjectId` value.

Note that you will see a different `ObjectId` value from this example because `ObjectId` values are specific to machine and time when the `insertOne()` method executes.

To select the document that you have inserted, you can use the `find()` (https://www.mongodbtutorial.org/mongodb-crud/mongodb-find/) method like this:

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

Output:

```
[
  {
    _id: ObjectId("621489fcf514a446bf1a98ea"),
    title: 'MongoDB insertOne',
    isbn: '0-7617-6154-3'
  }
]
```

2) Insert a document with an _id field example

The following example uses the `insertOne()` method to insert a document that has an `_id` field into the `books` collection:

```
db.books.insertOne({
   _id: 1,
   title: "Mastering Big Data",
   isbn: "0-9270-4986-4"
});
```

Output:

```
{ "acknowledged" : true, "insertedId" : 1 }
```

The following example attempts to insert another document whose `_id` field already exists into the `books` collection:

```
db.books.insertOne({
   _id: 1,
   title: "MongoDB for JS Developers",
   isbn: "0-4925-3790-9"
});
```

Since the `_id: 1` already exists, MongoDB threw the following exception:

```
WriteError({
        "index" : 0,
        "code" : 11000,
        "errmsg" : "E11000 duplicate key error collection: bookstore.books ind
        "op" : {
                "_id" : 1,
                "title" : "MongoDB for JS Developers",
                "isbn" : "0-4925-3790-9"
        }
})
```

Summary

- Use `db.collection.insertOne()` method to insert a single document into a collection.
- If you explicitly provide a value for the `_id` field, you must ensure that the value is unique within the collection or you will get a duplicate key error.

MongoDB insertMany

Summary: in this tutorial, you'll learn how to use the MongoDB `insertMany()` method to insert multiple documents into a collection.

Introduction to the MongoDB insertMany() method

The `insertMany()` allows you to insert multiple documents into a collection. Here is the syntax of the `insertMany()` method:

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

The `insertMany()` method accepts two arguments:

[document1, document2, ...]

The first argument is an array of documents that you want to insert into the collection.

Option

The second argument is a document that contains two optional field-and-value pairs:

```
{
   writeConcern: <document>,
   ordered: <boolean>
}
```

The `writeConcern` specifies the write concern. If you omit it, the `insertMany()` method will use the default write concern.

The `ordered` is a boolean value that determines whether MongoDB should perform an ordered or unordered insert.

When the `ordered` is set to `true`, the `insertMany()` method inserts documents in order. This is also the default option.

If the `ordered` is set to `false`, MongoDB may reorder the documents before inserts to increase performance. Therefore, you should not depend on the ordering of inserts if the `ordered` is set to `false`. You'll see how the `ordered` affects the behaviors of the insert in the example section.

The `insertMany()` method returns a document that contains:

- The `acknowledged` key sets to `true` if operation executed with a write concern or `false` if the write concern was disabled.
- An array of `_id` values of successfully inserted documents.

Collection creation

If the collection doesn't exist, the `insertMany()` method will create the collection and insert the documents. And it only creates the collection when the insert operation is successful.

_id field

If you don't specify the `_id` field for the document, the MongoDB generates a unique `ObjectId` value, assigns it to the `_id` field, and adds the `_id` field to the document before insert.

If you specify the `_id` fields for the document, it must be unique within the collection or you'll get a duplicate key error.

Error handling

The `insertMany()` throws a `BulkWriteError` exception in case of an error.

If an error occurs, the ordered insert will stop while the unordered insert will continue to process for the remaining documents in the queue.

MongoDB insertMany() method examples

First, launch `mongo` shell from the Terminal on macOS and Linux or Command Prompt on Windows and connect to the `bookdb` database on the local MongoDB server

```
mongosh bookdb
```

1) Using MongoDB insertMany() method to insert multiple documents without specifying _id fields

The following statement uses the `insertMany()` method to insert multiple documents without the `_id` fields:

```
db.books.insertMany([
   { title:  "NoSQL Distilled", isbn: "0-4696-7030-4"},
   { title:  "NoSQL in 7 Days", isbn: "0-4086-6859-8"},
   { title:  "NoSQL Database", isbn: "0-2504-6932-4"},
]);
```

Output:

```
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId("62148d16f514a446bf1a98f1"),
    '1': ObjectId("62148d16f514a446bf1a98f2"),
    '2': ObjectId("62148d16f514a446bf1a98f3")
  }
}
```

Because we did not specify the `_id` fields for the documents, MongoDB added the `_id` field with unique `ObjectId` to each document.

To retrieve the inserted documents, you use the `find()` method like this:

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

Output:

```
[
  {
    _id: ObjectId("62148d16f514a446bf1a98f1"),
    title: 'NoSQL Distilled',
    isbn: '0-4696-7030-4'
  },
  {
    _id: ObjectId("62148d16f514a446bf1a98f2"),
    title: 'NoSQL in 7 Days',
    isbn: '0-4086-6859-8'
  },
  {
    _id: ObjectId("62148d16f514a446bf1a98f3"),
    title: 'NoSQL Database',
    isbn: '0-2504-6932-4'
  }
]
boo
```

2) Using MongoDB insertMany() method to insert multiple documents with _id fields

The following statement uses the `insertMany()` method to insert multiple documents with the `_id` fields:

```
db.books.insertMany([
   { _id: 1, title:  "SQL Basics", isbn: "0-7925-6962-8"},
   { _id: 2, title:  "SQL Advanced", isbn: "0-1184-7778-1"}
]);
```

Output:

```
{ acknowledged: true, insertedIds: { '0': 1, '1': 2 } }
```

The following statement attempts to insert documents whose `_id` value already exist:

```
db.books.insertMany([
   { _id: 2, title:  "SQL Performance Tuning", isbn: "0-6799-2974-6"},
   { _id: 3, title:  "SQL Index", isbn: "0-5097-1723-3"}
]);
```

Since the `_id: 2` already exists, MongoDB threw the following exception:

```
Uncaught:
MongoBulkWriteError: E11000 duplicate key error collection: bookdb.books index
Result: BulkWriteResult {
  result: {
    ok: 1,
    writeErrors: [
      WriteError {
        err: {
          index: 0,
          code: 11000,
          errmsg: 'E11000 duplicate key error collection: bookdb.books index:
          errInfo: undefined,
          op: {
            _id: 2,
            title: 'SQL Performance Tuning',
            isbn: '0-6799-2974-6'
          }
        }
      }
    ],
    writeConcernErrors: [],
    insertedIds: [ { index: 0, _id: 2 }, { index: 1, _id: 3 } ],
```

```
      nInserted: 0,
      nUpserted: 0,
      nMatched: 0,
      nModified: 0,
      nRemoved: 0,
      upserted: []
    }
}
```

3) Unordered insert example

The following example uses the `insertMany()` method to perform an unordered insert:

```
db.books.insertMany(
   [{ _id: 3, title:  "SQL Performance Tuning", isbn: "0-6799-2974-6"},
   { _id: 3, title:  "SQL Trees", isbn: "0-6998-1556-8"},
   { _id: 4, title:  "SQL Graph", isbn: "0-6426-4996-0"},
   { _id: 5, title:  "NoSQL Pros", isbn: "0-9602-9886-X"}],
   { ordered: false }
);
```

In this example, the `_id: 3` is duplicated, MongoDB threw an error.

Since this example used the unordered insert, the operation continued to insert the documents with `_id` 4 and 5 into the `books` collection.

The following statement retrieves the inserted documents:

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

Output:

```
[
  {
    _id: ObjectId("62148d16f514a446bf1a98f1"),
```

```
    title: 'NoSQL Distilled',
    isbn: '0-4696-7030-4'
  },
  {
    _id: ObjectId("62148d16f514a446bf1a98f2"),
    title: 'NoSQL in 7 Days',
    isbn: '0-4086-6859-8'
  },
  {
    _id: ObjectId("62148d16f514a446bf1a98f3"),
    title: 'NoSQL Database',
    isbn: '0-2504-6932-4'
  },
  { _id: 1, title: 'SQL Basics', isbn: '0-7925-6962-8' },
  { _id: 2, title: 'SQL Advanced', isbn: '0-1184-7778-1' },
  { _id: 3, title: 'SQL Performance Tuning', isbn: '0-6799-2974-6' },
  { _id: 4, title: 'SQL Graph', isbn: '0-6426-4996-0' },
  { _id: 5, title: 'NoSQL Pros', isbn: '0-9602-9886-X' }
]
```

Summary

- Use the `db.collection.insertMany()` method to insert multiple documents into a collection.
- When the `ordered` is `true`, the `insertMany()` performs an unordered insert; otherwise, it performs an unordered insert.

MongoDB findOne

Summary: in this tutorial, you'll learn how to use the MongoDB `findOne()` method to retrieve a single document from a collection.

Introduction to MongoDB findOne() method

The `findOne()` returns a single document from a collection that satisfies the specified condition.

The `findOne()` method has the following syntax:

```
db.collection.findOne(query, projection)
```

The `findOne()` accepts two optional arguments: `query` and `projection` .

- The `query` is a document that specifies the selection criteria.
- The `projection` is a document that specifies the fields in the matching document that you want to return.

If you omit the `query` , the `findOne()` returns the first document in the collection according to the natural order which is the order of documents on the disk.

If you don't pass the `projection` argument, then `findOne()` will include all fields in the matching documents.

To specify whether a field should be included in the returned document, you use the following format:

```
{field1: value, field1: value, ... }
```

If the `value` is `true` or `1` , MongoDB will include the field in the returned document. In case the `value` is `false` or `0` , MongoDB won't include it.

By default, MongoDB always includes the `_id` field in the returned documents. To suppress it, you need to explicitly specify `_id: 0` in the `projection` argument.

If multiple documents satisfy the `query`, the `findOne()` method returns the first document based on the order of documents stored on the data storage.

> Note that there are other forms of projections such as array projection and aggregation projection which are not covered in this tutorial.

MongoDB findOne() method examples

We'll use the following `products` collection for the demonstration:

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate": ISODate("201
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate": ISODate("20
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate": ISODate
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate": ISODate("2
    { "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate": ISODate("
])
```

1) Using MongoDB findOne() method with zero argument

The following example uses the `findOne()` method to select the first document from the `products` collection:

```
db.products.findOne()
```

The statement returns all fields of the matching document:

```
{
  _id: 1,
  name: 'xPhone',
  price: 799,
```

```
    releaseDate: ISODate("2011-05-14T00:00:00.000Z"),
    spec: { ram: 4, screen: 6.5, cpu: 2.66 },
    color: [ 'white', 'black' ],
    storage: [ 64, 128, 256 ]
  }
```

Note that omitting the `query` is the same as passing the `query` as an empty document:

```
db.products.findOne({})
```

2) Using MongoDB findOne() method with a filter

The following statement uses the `findOne()` method to find the document whose `_id` is 2.

```
db.products.findOne({_id:2})
```

It returns all the fields of the document with `_id` 2:

```
{
  _id: 2,
  name: 'xTablet',
  price: 899,
  releaseDate: ISODate("2011-09-01T00:00:00.000Z"),
  spec: { ram: 16, screen: 9.5, cpu: 3.66 },
  color: [ 'white', 'black', 'purple' ],
  storage: [ 128, 256, 512 ]
}
```

3) Using MongoDB findOne() method to select some fields

The following example uses the `findOne()` method to find the document with `_id 5` . And it returns only the `_id` and `name` fields of the matching document:

```
db.products.findOne({_id: 5}, {name: 1})
```

The query returned the following document:

```
{ "_id" : 5, "name" : "SmartPhone" }
```

As you can see clearly from the output, MongoDB includes the `_id` field in the returned document by default.

To completely remove it from the returned document, you need to explicitly specify `_id:0` in the `projection` document like this:

```
db.products.findOne({ _id: 5 }, {name: 1, _id: 0})
```

It returned the following document:

```
{ "name" : "SmartPhone" }
```

Summary

- Use the `findOne()` method to retrieve the first document in a collection that satisfies the specified condition.

MongoDB find

Summary: in this tutorial, you'll learn how to use the MongoDB `find()` method to select documents from a collection.

Introduction to the MongoDB find() method

The `find()` method finds the documents that satisfy a specified condition and returns a cursor to the matching documents.

The following shows the syntax of the `find()` method:

```
db.collection.find(query, projection)
```

Similar to the `findOne()` (https://www.mongodbtutorial.org/mongodb-crud/mongodb-findone/) method, the `find()` method accepts two optional arguments.

1) query

The `query` is a document that specifies the criteria for selecting documents from the collection. If you omit the `query` or pass an empty document(`{}`), the `find()` returns a cursor that returns all documents in the collection.

2) projection

The `projection` is a document that specifies the fields in the matching documents to return. If you omit the `projection` argument, the `find()` method returns all fields in the matching documents.

By default, the `find()` method includes the `_id` field in the matching documents unless you explicitly specify `_id: false` in the `projection` document.

Since the mongo shell automatically iterates the cursor returned by the `find()` method, you don't need to do any extra steps to get the document from the cursor.

By default, the mongo shell shows up the first 20 documents. To continue iteration, you type the `it` command in the shell.

The MongoDB find() method examples

We'll use the following `books` collection for the demonstration:

```
db.books.insertMany([
        { "_id" : 1, "title" : "Unlocking Android", "isbn" : "1933988673", "ca
        { "_id" : 2, "title" : "Android in Action, Second Edition", "isbn" : "
        { "_id" : 3, "title" : "Specification by Example", "isbn" : "161729008
        { "_id" : 4, "title" : "Flex 3 in Action", "isbn" : "1933988746", "cat
        { "_id" : 5, "title" : "Flex 4 in Action", "isbn" : "1935182420", "cat
        { "_id" : 6, "title" : "Collective Intelligence in Action", "isbn" : "
        { "_id" : 7, "title" : "Zend Framework in Action", "isbn" : "193398832
        { "_id" : 8, "title" : "Flex on Java", "isbn" : "1933988797", "categor
        { "_id" : 9, "title" : "Griffon in Action", "isbn" : "1935182234", "ca
        { "_id" : 10, "title" : "OSGi in Depth", "isbn" : "193518217X", "categ
        { "_id" : 11, "title" : "Flexible Rails", "isbn" : "1933988509", "cate
        { "_id" : 13, "title" : "Hello! Flex 4", "isbn" : "1933988762", "categ
        { "_id" : 14, "title" : "Coffeehouse", "isbn" : "1884777384", "categor
        { "_id" : 15, "title" : "Team Foundation Server 2008 in Action", "isbn
        { "_id" : 16, "title" : "Brownfield Application Development in .NET",
        { "_id" : 17, "title" : "MongoDB in Action", "isbn" : "1935182870", "c
        { "_id" : 18, "title" : "Distributed Application Development with Powe
        { "_id" : 19, "title" : "Jaguar Development with PowerBuilder 7", "isb
        { "_id" : 20, "title" : "Taming Jaguar", "isbn" : "1884777686", "categ
        { "_id" : 21, "title" : "3D User Interfaces with Java 3D", "isbn" : "1
        { "_id" : 22, "title" : "Hibernate in Action", "isbn" : "193239415X",
        { "_id" : 23, "title" : "Hibernate in Action (Chinese Edition)", "cate
        { "_id" : 24, "title" : "Java Persistence with Hibernate", "isbn" : "1
        { "_id" : 25, "title" : "JSTL in Action", "isbn" : "1930110529", "cate
        { "_id" : 26, "title" : "iBATIS in Action", "isbn" : "1932394826", "ca
        { "_id" : 27, "title" : "Designing Hard Software", "isbn" : "133046192
        { "_id" : 28, "title" : "Hibernate Search in Action", "isbn" : "193398
        { "_id" : 29, "title" : "jQuery in Action", "isbn" : "1933988355", "ca
```

```
        { "_id" : 30, "title" : "jQuery in Action, Second Edition", "isbn" : "
]);
```

1) Using MongoDB find() method to retrieve all documents from a collection

The following example uses the `find()` method with no parameters to return all documents from the `books` collection:

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

In the mongo shell, the statement returns the first 20 documents with all available fields in the matching documents.

If you type `it` command and press enter, you'll see the next 20 documents.

2) Using MongoDB find() method to search for a specific document

The following example uses the `find()` method to search for the document with id 10:

```
db.books.find({_id: 10})
```

The statement returns the document whose `_id` is 10. Since it doesn't have the `projection` argument, the returned document includes all available fields:

```
[
  {
    _id: 10,
    title: 'OSGi in Depth',
    isbn: '193518217X',
    categories: [ 'Java' ]
  }
]
```

3) Using MongoDB find() method to return selected fields

The following example uses the `find()` method to search for documents whose category is `Java`. It returns the fields `_id`, `title` and `isbn`:

```
db.books.find({ categories: 'Java'}, { title: 1,isbn: 1})
```

Output:

```
[
  {
    _id: 2,
    title: 'Android in Action, Second Edition',
    isbn: '1935182722'
  },
  { _id: 9, title: 'Griffon in Action', isbn: '1935182234' },
  { _id: 10, title: 'OSGi in Depth', isbn: '193518217X' },
  {
    _id: 21,
    title: '3D User Interfaces with Java 3D',
    isbn: '1884777902'
  },
  { _id: 22, title: 'Hibernate in Action', isbn: '193239415X' },
  { _id: 23, title: 'Hibernate in Action (Chinese Edition)' },
  {
    _id: 24,
    title: 'Java Persistence with Hibernate',
    isbn: '1932394885'
  },
  { _id: 28, title: 'Hibernate Search in Action', isbn: '1933988649' },
  {
    _id: 30,
    title: 'jQuery in Action, Second Edition',
    isbn: '1935182323'
  }
]
```

To remove the `_id` field from the matching documents, you need to explicitly specify `_id: 0` in the `projection` argument like this:

```
db.books.find({ categories: 'Java'}, { title: 1,isbn: 1,_id: 0})
```

Output:

```
[
  { title: 'Android in Action, Second Edition', isbn: '1935182722' },
  { title: 'Griffon in Action', isbn: '1935182234' },
  { title: 'OSGi in Depth', isbn: '193518217X' },
  { title: '3D User Interfaces with Java 3D', isbn: '1884777902' },
  { title: 'Hibernate in Action', isbn: '193239415X' },
  { title: 'Hibernate in Action (Chinese Edition)' },
  { title: 'Java Persistence with Hibernate', isbn: '1932394885' },
  { title: 'Hibernate Search in Action', isbn: '1933988649' },
  { title: 'jQuery in Action, Second Edition', isbn: '1935182323' }
]
```

> Note that you'll learn how to construct more complex conditions using operators in the next tutorials.

Summary

- Use the `find()` method to select the documents from a collection and returns a cursor referencing the matching documents.

MongoDB Projection

Summary: in this tutorial, you'll how to use the MongoDB projection that allows you to select fields to return from a query

Introduction to the MongoDB projection

In MongoDB, projection simply means selecting fields to return from a query.

By default, the `find()` (https://www.mongodbtutorial.org/mongodb-crud/mongodb-find/) and `findOne()` (https://www.mongodbtutorial.org/mongodb-crud/mongodb-findone/) methods return all fields in matching documents. Most of the time you don't need data from all the fields.

To select fields to return from a query, you can specify them in a document and pass the document to the `find()` and `findOne()` methods. This document is called a projection document.

To determine if a field is included in the returned documents, you use the following syntax:

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

If the `value` is `1` or `true`, the `<field>` is included in the matching documents. In case the `value` is `0` or `false`, it is suppressed from the returned documents.

If the `projection` document is empty `{}`, all the available fields will be included in the returned documents.

To specify a field in an embedded document, you use the following dot notation:

```
{ "<embeddedDocument>.<field>": value, ... }
```

Similarly, to include a `<field>` from an embedded document located in an array, you use the following dot notation syntax:

```
{ "<arrayField>.field": value, ...}
```

MongoDB projection examples

We'll use the following `products` collection for the projection examples.

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate": ISODate("201
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate": ISODate("20
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate": ISODate
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate": ISODate("2
    { "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate": ISODate("
 ])
```

1) Returning all fields in matching documents

If you don't specify the projection document, the `find()` method will include all fields in the matching documents.

For example, the following query returns all fields from all documents in the `products` collection where the `price` is `899` :

```
db.products.find({price: 899});
```

Output:

```
[
  {
    _id: 2,
    name: 'xTablet',
    price: 899,
    releaseDate: ISODate("2011-09-01T00:00:00.000Z"),
    spec: { ram: 16, screen: 9.5, cpu: 3.66 },
    color: [ 'white', 'black', 'purple' ],
```

```
    storage: [ 128, 256, 512 ],
    inventory: [ { qty: 300, warehouse: 'San Francisco' } ]
  },
  {
    _id: 3,
    name: 'SmartTablet',
    price: 899,
    releaseDate: ISODate("2015-01-14T00:00:00.000Z"),
    spec: { ram: 12, screen: 9.7, cpu: 3.66 },
    color: [ 'blue' ],
    storage: [ 16, 64, 128 ],
    inventory: [
      { qty: 400, warehouse: 'San Jose' },
      { qty: 200, warehouse: 'San Francisco' }
    ]
  }
]
```

2) Returning specified fields including the _id field

If you specify the fields in the projection document, the `find()` method will return only these fields including the `_id` field by default.

The following example returns all documents from the `products` collection. However, its result includes only the `name`, `price`, and `_id` field in the matching documents:

```
db.products.find({}, {
    name: 1,
    price: 1
});
```

Output:

```
[
  { _id: 1, name: 'xPhone', price: 799 },
  { _id: 2, name: 'xTablet', price: 899 },
```

```
    { _id: 3, name: 'SmartTablet', price: 899 },
    { _id: 4, name: 'SmartPad', price: 699 },
    { _id: 5, name: 'SmartPhone', price: 599 }
]
```

To suppress the `_id` field, you need to explicitly specify it in the projection document like this:

```
db.products.find({}, {
    name: 1,
    price: 1
,
    _id: 0
});
```

Output:

```
[
  { name: 'xPhone', price: 799 },
  { name: 'xTablet', price: 899 },
  { name: 'SmartTablet', price: 899 },
  { name: 'SmartPad', price: 699 },
  { name: 'SmartPhone', price: 599 }
]
```

3) Returning all fields except for some fields

If the number of fields to return is many, you can use the projection document to exclude other fields instead.

The following example returns all fields of the document `_id 1` except for `releaseDate` , `spec` , and `storage` fields:

```
db.products.find({_id:1}, {
    releaseDate: 0,
    spec: 0,
```

```
    storage: 0
})
```

Output:

```
[
  {
    _id: 1,
    name: 'xPhone',
    price: 799,
    color: [ 'white', 'black' ],
    inventory: [ { qty: 1200, warehouse: 'San Jose' } ]
  }
]
```

4) Returning fields in embedded documents

The following example returns the `name`, `price`, and `_id` fields of document `_id 1`. It also returns the `screen` field on the `spec` embedded document:

```
db.products.find({_id:1}, {
    name: 1,
    price: 1,
    "spec.screen": 1
})
```

Output:

```
[ { _id: 1, name: 'xPhone', price: 799, spec: { screen: 6.5 } } ]
```

MongoDB 4.4 and later allows you to specify embedded fields using the nested form like this:

```
db.products.find({_id:1}, {
    name: 1,
    price: 1,
```

```
    spec : { screen: 1 }
})
```

5) Projecting fields on embedded documents in an array

The following example specifies a projection that returns:

- the `_id` field (by default)
- The `name` field
- And `qty` field in the documents embedded in the `inventory` array.

```
db.products.find({}, {
    name: 1,
    "inventory.qty": 1
});
```

Output:

```
[
  { _id: 1, name: 'xPhone', inventory: [ { qty: 1200 } ] },
  { _id: 2, name: 'xTablet', inventory: [ { qty: 300 } ] },
  {
    _id: 3, name: 'SmartTablet', inventory: [ { qty: 400 }, { qty: 200 } ]
  },
  { _id: 4, name: 'SmartPad', inventory: [ { qty: 1200 } ] },
  { _id: 5, name: 'SmartPhone' }
]
```

Summary

- Use the `{<field>: 1}` to include the `<field>` in the matching documents and `{<field>: 0}` to exclude it.
- Use the `{ <embeddedDocument>.<field>: 1}` to include the `<field>` from the `<embeddedDocument>` in the matching document and `{ <embeddedDocument>.<field>: 0}` to suppress it.

- Use `{ <arrayField>.<field>: 1}` to include the `<field>` from the embedded document in an array in the matching document and `{ <arrayField>.<field>: 0}` to exclude it.

MongoDB $eq

Summary: in this tutorial, you'll learn how to use the MongoDB `$eq` operator to specify an equality condition.

Introduction to the MongoDB $eq operator

The `$eq` operator is a comparison query operator that allows you to match documents where the value of a field equals a specified value.

The following shows the syntax of `$eq` operator:

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

The query is equivalent to the following:

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

MongoDB $eq operator examples

We'll use the following `products` collection for the demonstration:

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate": ISODate("201
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate": ISODate("20
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate": ISODate
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate": ISODate("2
    { "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate": ISODate("
 ])
```

1) Using $eq operator to check if a field equals a specified value

The following example uses the `$eq` operator to query the `products` collection to select all documents where the value of the `price` field equals `899`:

```
db.products.find({
    price: {
        $eq: 899
    }
}, {
    name: 1,
    price: 1
})
```

The query is equivalent to the following:

```
db.products.find({
    price: 899
}, {
    name: 1,
    price: 1
})
```

They both match the following documents:

```
[
  { _id: 2, name: 'xTablet', price: 899 },
  { _id: 3, name: 'SmartTablet', price: 899 }
]
```

2) Using the $eq operator to check if a field in an embedded document equals a value

The following example uses the `$eq` operator to search for documents where the value of the `ram` field in the `spec` document equals `4`:

```
db.products.find({
    "spec.ram": {
        $eq: 4
    }
}, {
    name: 1,
    "spec.ram": 1
})
```

It is equivalent to the following:

```
db.products.find({
    "spec.ram": 4

}, {
    name: 1,
    "spec.ram": 1
})
```

Both of these queries returns the following documents:

```
[
  { _id: 1, name: 'xPhone', spec: { ram: 4 } },
  { _id: 5, name: 'SmartPhone', spec: { ram: 4 } }
]
```

3) Using $eq operator to check if an array element equals a value

The following example uses the `$eq` operator to query the `products` collection to find all documents where the array `color` contains an element with the value `"black"` :

```
db.products.find({
    color: {
        $eq: "black"
    }
```

```
}, {
    name: 1,
    color: 1
})
```

It's equivalent to:

```
db.products.find({
    color: "black"

}, {
    name: 1,
    color: 1
})
```

Both queries return the following matching documents:

```
[
  { _id: 1, name: 'xPhone', color: [ 'white', 'black' ] },
  { _id: 2, name: 'xTablet', color: [ 'white', 'black', 'purple' ] }
]
```

4) Using $eq operator to check if a field equals a date

The following example uses the `$eq` operator to select documents in the `widget` collection with the published date is `2020-05-14` :

```
db.products.find({
    releaseDate: {
        $eq: new ISODate("2020-05-14")
    }
}, {
    name: 1,
    releaseDate: 1
})
```

It returned the following document:

```
[
  {
    _id: 4,
    name: 'SmartPad',
    releaseDate: ISODate("2020-05-14T00:00:00.000Z")
  }
]
```

Summary

- Use the `$eq` operator to specify an equality condition.

MongoDB $gt

Summary: in this tutorial, you will learn how to use the MongoDB `$gt` operator to select documents where the value of a field is greater than a specified value.

Introduction to the MongoDB $gt operator

The `$gt` operator is a comparison query operator that allows you to select documents where the value of a field is greater than (`>`) a specified value.

The following shows the syntax of the `$gt` operator:

```
{ field: { $gt: value}}
```

MongoDB $gt operator example

We'll use the following `widget` collection:

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate": ISODate("201
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate": ISODate("20
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate": ISODate
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate": ISODate("2
    { "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate": ISODate("
 ])
```

1) Using $gt to select documents where the value of a field is greater than a specified value

The following example uses the `$gt` operator to select documents from the `products` collection where `price` is greater than `699` :

```
db.products.find({
    price: {
        $gt: 699
    }
}, {
    name: 1,
    price: 1
})
```

The query returned the following documents:

```
[
  { _id: 1, name: 'xPhone', price: 799 },
  { _id: 2, name: 'xTablet', price: 899 },
  { _id: 3, name: 'SmartTablet', price: 899 }
]
```

2) Using the $gt operator to check if the value of a field in an embedded document is greater than a value

The following example uses `$gt` operator to select documents where the value of the `ram` field in the `spec` document is greater than `8` :

```
db.products.find({
    "spec.ram": {
        $gt: 8
    }
}, {
    name: 1,
    "spec.ram": 1
});
```

Output:

```
[
  { _id: 2, name: 'xTablet', spec: { ram: 16 } },
  { _id: 3, name: 'SmartTablet', spec: { ram: 12 } }
]
```

3) Using the $gt operator to check if an array element is greater than a value

The following example uses the `$gt` operator to query the `products` collection to find all documents where the `storage` array has at least one element greater than 128:

```
db.products.find({
    storage: {
        $gt: 128
    }
}, {
    name: 1,
    storage: 1
})
```

The query returned the following documents:

```
[
  { _id: 1, name: 'xPhone', storage: [ 64, 128, 256 ] },
  { _id: 2, name: 'xTablet', storage: [ 128, 256, 512 ] },
  { _id: 4, name: 'SmartPad', storage: [ 128, 256, 1024 ] },
  { _id: 5, name: 'SmartPhone', storage: [ 128, 256 ] }
]
```

4) Using the $gt operator to check if the value of a field is after a date

The following example uses the `$gt` operator to query documents from the `products` collection to find all documents where the release date is after `2015-01-01` :

```
db.products.find({
    "releaseDate": {
        $gt: new ISODate('2015-01-01')
    }
}, {
    name: 1,
    releaseDate: 1
});
```

The query returned the following documents:

```
[
  {
    _id: 3,
    name: 'SmartTablet',
    releaseDate: ISODate("2015-01-14T00:00:00.000Z")
  },
  {
    _id: 4,
    name: 'SmartPad',
    releaseDate: ISODate("2020-05-14T00:00:00.000Z")
  },
  {
    _id: 5,
    name: 'SmartPhone',
    releaseDate: ISODate("2022-09-14T00:00:00.000Z")
  }
]
```

Summary

- Use the `$gt` operator to select documents where a field is greater than a specified value.

MongoDB $gte

Summary: in this tutorial, you will learn how to use the MongoDB `$gte` operator to select documents where the value of a field is greater than or equal to a specified value.

Introduction to the MongoDB $gte operator

The `$gte` is a comparison query operator that allows you to select documents where a value of a field is greater than or equal to (i.e. >=) a specified value.

The `$gte` operator has the following syntax:

```
{field: {$gte: value} }
```

MongDB $gte operator examples

We'll use the following `products` collection:

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate": ISODate("201
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate": ISODate("20
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate": ISODate
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate": ISODate("2
    { "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate": ISODate("
 ])
```

1) Using $gte operator to select documents where a field is greater than or equal to a specified value

The following example uses the `$gt` e operator to select documents from the `products` collection where `price` is greater than 799:

```
db.products.find({
    price: {
        $gte: 799
    }
}, {
    name: 1,
    price: 1
})
```

The query returned the following documents:

```
[
  { _id: 1, name: 'xPhone', price: 799 },
  { _id: 2, name: 'xTablet', price: 899 },
  { _id: 3, name: 'SmartTablet', price: 899 }
]
```

2) Using the $gte operator to check if a field in an embedded document is greater than or equal to a value

The following query uses `$gte` operator to select documents where the value of the `screen` field in the `spec` document is greater than or equal to `9.5` :

```
db.products.find({
    "spec.screen": {
        $gte: 9.5
    }
}, {
    name: 1,
    "spec.screen": 1
})
```

Output:

```
[
  { _id: 2, name: 'xTablet', spec: { screen: 9.5 } },
  { _id: 3, name: 'SmartTablet', spec: { screen: 9.7 } },
  { _id: 4, name: 'SmartPad', spec: { screen: 9.7 } },
  { _id: 5, name: 'SmartPhone', spec: { screen: 9.7 } }
]
```

3) Using the $gte operator to check if an array element is greater than or equal to a value

The following example uses the `$gte` operator to query the `products` collection to find all documents where the array `storage` has at least one element greater than or equal to 512:

```
db.products.find({
    storage: {
        $gte: 512
    }
}, {
    name: 1,
    storage: 1
})
```

The query returned the following documents:

```
[
  { _id: 2, name: 'xTablet', storage: [ 128, 256, 512 ] },
  { _id: 4, name: 'SmartPad', storage: [ 128, 256, 1024 ] }
]
```

4) Using the $gte operator to check if a field is after or on the same date

The following query uses the `$gte` operator to select documents from the `products` collection to find (https://www.mongodbtutorial.org/mongodb-crud/mongodb-find/) all documents where the release date

is after or on `2020-05-14` :

```
db.products.find({
    "releaseDate": {
        $gte: new ISODate('2020-05-14')
    }
}, {
    name: 1,
    releaseDate: 1
});
```

The query returned the following documents:

```
[
  {
    _id: 4,
    name: 'SmartPad',
    releaseDate: ISODate("2020-05-14T00:00:00.000Z")
  },
  {
    _id: 5,
    name: 'SmartPhone',
    releaseDate: ISODate("2022-09-14T00:00:00.000Z")
  }
]
```

Summary

- Use the `$gt` e operator to select documents where a field is greater than or equal to a specified value.

MongoDB $lt

Summary: in this tutorial, you will learn how to use the MongoDB $lt operator to select documents where the value of a field is less than (`<`) a specified value.

Introduction to the MongoDB $lt operator

The `$lt` operator is a comparison query operator that allows you to select the documents where the value of a field is less than a specified value.

Here is the syntax of the $lt operator:

```
{field: {$lt: value} }
```

MongoDB $lt operator examples

We'll use the following `products` collection:

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate": ISODate("201
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate": ISODate("20
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate": ISODate
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate": ISODate("2
    { "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate": ISODate("
 ])
```

1) Using $lt operator to select documents where a field is less than a specified value

The following example uses the `$lt` operator to select documents from the `products` collection where `price` is less than 799:

```
db.products.find({
    price: {
        $lt: 799
    }
}, {
    name: 1,
    price: 1
})
```

Output:

```
[
  { _id: 4, name: 'SmartPad', price: 699 },
  { _id: 5, name: 'SmartPhone', price: 599 }
]
```

2) Using the $lt operator to check if a field in an embedded document is less than a value

The following query uses `$lt` operator to select documents where the value of the `screen` field in the `spec` document is less than `7` :

```
db.products.find({
    "spec.screen": {
        $lt: 7
    }
}, {
    name: 1,
    "spec.screen": 1
})
```

Output:

```
[ { _id: 1, name: 'xPhone', spec: { screen: 6.5 } } ]
```

3) Using the $lt operator to check if an array element is greater than or equal to a value

The following example uses the `$lt` operator to query the `products` collection to find all documents where the array `storage` has at least one element less than `128`:

```
db.products.find({
    storage: {
        $lt: 128
    }
}, {
    name: 1,
    storage: 1
})
```

The query returned the following documents:

```
[
  { _id: 1, name: 'xPhone', storage: [ 64, 128, 256 ] },
  { _id: 3, name: 'SmartTablet', storage: [ 16, 64, 128 ] }
]
```

4) Using the $lt operator to check if a field is before a date

The following query uses the `$lt` operator to select documents from the `products` collection to find all documents where the release date before `2015-01-01`:

```
db.products.find({
    "releaseDate": {
        $lt: new ISODate('2015-01-01')
    }
}, {
```

```
    name: 1,
    releaseDate: 1
})
```

The query returned the following documents:

```
[
  {
    _id: 1,
    name: 'xPhone',
    releaseDate: ISODate("2011-05-14T00:00:00.000Z")
  },
  {
    _id: 2,
    name: 'xTablet',
    releaseDate: ISODate("2011-09-01T00:00:00.000Z")
  }
]
```

Summary

- Use the `$lt` operator to select documents where a field is less a specified value.

MongoDB $lte

Summary: in this tutorial, you will learn how to use the MongoDB `$lte` operator to select documents where the value of a field is less than or equal to a specified value.

Introduction to the MongoDB $lte operator

The `$lte` is a comparison query operator that allows you to select documents where the value of a field is less than or equal to (`<=`) a specified value.

The following shows the `$lte` syntax:

```
{field: {$lte: value} }
```

MongDB $lte operator examples

We'll use the following `products` collection:

```
db.products.drop();
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate": ISODate("201
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate": ISODate("20
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate": ISODate
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate": ISODate("2
    { "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate": ISODate("
 ]);
```

1) Using $lte operator to select documents where the value of a field is less than or equal to a specified value

The following example uses the `$gt` e operator to select documents from the `products` collection where `price` is less than 799:

```
db.products.find({
    price: {
        $lte: 799
    }
}, {
    name: 1,
    price: 1
})
```

Output:

```
[
  { _id: 1, name: 'xPhone', price: 799 },
  { _id: 4, name: 'SmartPad', price: 699 },
  { _id: 5, name: 'SmartPhone', price: 599 }
]
```

2) Using the $lte operator to check if the value of a field in an embedded document is less than or equal to a value

The following query uses `$lte` operator to select documents where the value of the `screen` field in the `spec` document is less than or equal to `6.5` :

```
db.products.find({
    "spec.screen": {
        $lte: 6.5
    }
}, {
    name: 1,
    "spec.screen": 1
})
```

Output:

```
[
  { _id: 1, name: 'xPhone', spec: { screen: 6.5 } },
  { _id: 5, name: 'SmartPhone', spec: { screen: 5.7 } }
]
```

3) Using the $lte operator to check if an array element is less than or equal to a value

The following example uses the `$lte` operator to query the `products` collection to find all documents where the array `storage` has at least one element less than or equal to 64:

```
db.products.find({
    storage: {
        $lte: 64
    }
}, {
    name: 1,
    storage: 1
})
```

The query returned the following documents:

```
[
  { _id: 1, name: 'xPhone', storage: [ 64, 128, 256 ] },
  { _id: 3, name: 'SmartTablet', storage: [ 16, 64, 128 ] }
]
```

4) Using the $lte operator to check if the value of a field is before or on the same date

The following query uses the `$lte` operator to select documents from the `products` collection to find all documents where the release date is before or on `2015-01-11` :

```
db.products.find({
    "releaseDate": {
        $lte: new ISODate('2015-01-01')
    }
}, {
    name: 1,
    releaseDate: 1
});
```

The query returned the following documents:

```
[
  {
    _id: 1,
    name: 'xPhone',
    releaseDate: ISODate("2011-05-14T00:00:00.000Z")
  },
  {
    _id: 2,
    name: 'xTablet',
    releaseDate: ISODate("2011-09-01T00:00:00.000Z")
  }
]
```

Summary

- Use the `$lte` operator to select documents where the value of a field is less than or equal to a specified value.

MongoDB $ne

Summary: in this tutorial, you'll learn how to use the MongoDB `$ne` operator to query documents from a collection.

Introduction to MongoDB $ne operator

The `$ne` is a comparison query operator that allows you to select documents where the value of a filed is **not equal to** a specified value. It also includes documents that **don't contain the field**.

The `$ne` is called the **inequality operator**. Here is the syntax of the `$ne` operator:

```
{ field: {$ne: value}}
```

MongoDB $ne operator examples

We'll use the following `products` collection:

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate": ISODate("201
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate": ISODate("20
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate": ISODate
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate": ISODate("2
    { "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate": ISODate("
,
    { "_id" : 6, "name" : "xWidget", "spec" : { "ram" : 64, "screen" : 9.7, "c
 ])
```

1) Using the $ne operator to select documents where the value of a field is greater than a specified value

The following example uses the `$ne` operator to select documents from the `products` collection where the price is not equal to `899`:

```
db.products.find({
    price: {
        $ne: 899
    }
}, {
    name: 1,
    price: 1
})
```

It matches the following documents:

```
[
  { _id: 1, name: 'xPhone', price: 799 },
  { _id: 4, name: 'SmartPad', price: 699 },
  { _id: 5, name: 'SmartPhone', price: 599 },
  { _id: 6, name: 'xWidget' }
]
```

2) Using the $ne operator to check if a field in an embedded document is not equal to a value

The following example uses `$ne` operator to select documents where the value of the `screen` field in the `spec` document is not equal to 9.7:

```
db.products.find({
    "spec.screen": {
        $ne: 9.7
    }
}, {
    name: 1,
    "spec.screen": 1
})
```

Output:

```
[
  { _id: 1, name: 'xPhone', spec: { screen: 6.5 } },
  { _id: 2, name: 'xTablet', spec: { screen: 9.5 } }
]
```

3) Using the $ne operator to check if an array element is not equal to a value

The following example uses the `$ne` operator to query the `products` collection to find documents where the array `storage` does not have any element that equals 128:

```
db.products.find({
    storage: {
        $ne: 128
    }
}, {
    name: 1,
    storage: 1
});
```

It matched the following documents:

```
[ { _id: 6, name: 'xWidget', storage: [ 1024 ] } ]
```

4) Using the $ne operator to check if the value of a field is not equal to a date

The following query uses the `$ne` operator to find documents from the `products` collection where the release date is not `2015-01-14` :

```
db.products.find({
    releaseDate: {
```

```
        $ne: new ISODate('2015-01-14')
    }
}, {
    name: 1,
    releaseDate: 1
});
```

It returns the documents whose release dates are not `2015-01-14` and also the document that does not include the field `releaseDate` :

```
[
  {
    _id: 1,
    name: 'xPhone',
    releaseDate: ISODate("2011-05-14T00:00:00.000Z")
  },
  {
    _id: 2,
    name: 'xTablet',
    releaseDate: ISODate("2011-09-01T00:00:00.000Z")
  },
  {
    _id: 4,
    name: 'SmartPad',
    releaseDate: ISODate("2020-05-14T00:00:00.000Z")
  },
  {
    _id: 5,
    name: 'SmartPhone',
    releaseDate: ISODate("2022-09-14T00:00:00.000Z")
  },
  { _id: 6, name: 'xWidget' }
]
```

Summary

- Use the `$ne` operator to check if the value of a field **is not equal** to a specified value.

MongoDB $in

Summary: in this tutorial, you'll learn how to use the MongoDB `$in` operator to select documents where the value of a field equals any value in an array.

Introduction to the MongoDB $in operator

The `$in` is a comparison query operator that allows you to select documents where the value of a field is equal to any value in an array.

The following shows the syntax of the `$in` operator:

```
{ field: { $in: [<value1>, <value2>,...] }}
```

If the `field` holds a single value, then the `$in` operator selects documents where the value of the `field` is equal to any value such as `<value1>`, `<value2>`.

In case the `field` holds an array, the `$in` operator selects documents where the array contains at least one element that equals any value (`<value1>`, `<value2>`).

The value list `<value1>`, `<value2>`, etc., can be a list of literal values or regular expressions.

> A regular expression (https://www.javascripttutorial.net/javascript-regular-expression/) is a set of characters that defines a search pattern e.g., `/\d+/` any digits such as 1, 123, and 1234.

MongoDB $in operator examples

We'll use this `products` collections in the following examples:

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate": ISODate("201
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate": ISODate("20
```

```
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate": ISODate
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate": ISODate("2
    { "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate": ISODate("
])
```

1) Using the $in opeator to match values

The following example uses the `$in` operator to select documents from the `products` collection whose the `price` is either `599` or `799` :

```
db.products.find({
    price: {
        $in: [699, 799]
    }
}, {
    name: 1,
    price: 1
})
```

It returned the following documents:

```
[
  { _id: 1, name: 'xPhone', price: 799 },
  { _id: 4, name: 'SmartPad', price: 699 }
]
```

2) Using the $in operator to match values in an array

The products collection has the `color` array that contains some colors.

The following example uses the `$in` operator to select documents where the `color` array has at least one element either `"black"` or `"white"` :

```
db.products.find({
    color: {
```

```
        $in: ["black", "white"]
    }
}, {
    name: 1,
    color: 1
})
```

The query returned the following documents:

```
[
  { _id: 1, name: 'xPhone', color: [ 'white', 'black' ] },
  { _id: 2, name: 'xTablet', color: [ 'white', 'black', 'purple' ] },
  {
    _id: 4,
    name: 'SmartPad',
    color: [ 'white', 'orange', 'gold', 'gray' ]
  },
  {
    _id: 5,
    name: 'SmartPhone',
    color: [ 'white', 'orange', 'gold', 'gray' ]
  }
]
```

3) Using the $in operator with regular expressions

The following query uses the $in operator to find documents where the `color` array has at least one element that matches either `/^g+/` or `/^w+/` regular expression:

```
db.products.find({
    color: {
        $in: [/^g+/, /^w+/]
    }
}, {
    name: 1,
```

```
    color: 1
})
```

It returned the following documents:

```
[
  { _id: 1, name: 'xPhone', color: [ 'white', 'black' ] },
  { _id: 2, name: 'xTablet', color: [ 'white', 'black', 'purple' ] },
  {
    _id: 4,
    name: 'SmartPad',
    color: [ 'white', 'orange', 'gold', 'gray' ]
  },
  {
    _id: 5,
    name: 'SmartPhone',
    color: [ 'white', 'orange', 'gold', 'gray' ]
  }
]
```

The `/^g+/` regular expression matches any string that begins with the letter `g` and is followed by any number of characters (`+`). Similarly, the `/^w+/` regular expression matches any string that starts with the letter `w` and is followed by any number of characters (`+`). This tutorial explains the regular expressions in JavaScript (https://www.javascripttutorial.net/javascript-regular-expression/) in detail.

Summary

- Use the MongoDB `$in` operator to select documents where the value of a field is equal to any values in an array.
- The values can be a list of literal values or regular expressions.

MongoDB $nin: Not In Operator

Summary: in this tutorial, you'll learn about the MongoDB $nin (Not In) operator and how to apply it effectively.

Introduction to the MongoDB $nin operator

The `$nin` is a query comparison operator that allows you to find (https://www.mongodbtutorial.org/mongodb-crud/mongodb-find/) documents where:

- the value of the field is not equal to (https://www.mongodbtutorial.org/mongodb-crud/mongodb-ne/) any value in an array
- or the field does not exist.

Here is the syntax of the `$nin` operator:

```
{ field: { $nin: [ <value1>, <value2> ...]} }
```

Like the `$in` (https://www.mongodbtutorial.org/mongodb-crud/mongodb-in/) operator, the value list (`<value1>` , `<value2>` ,...) can be a list of literal values or regular expressions.

MongoDB $nin operator examples

We'll use this `products` collections:

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate": ISODate("201
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate": ISODate("20
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate": ISODate
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate": ISODate("2
    { "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate": ISODate("
])
```

1) Using the MongoDB $nin opeator to match values

The following query uses the `$nin` operator to select documents from the `products` collection whose `price` is neither `599` or `799`:

```
db.products.find({
    price: {
        $nin: [699, 799]
    }
}, {
    name: 1,
    price: 1
})
```

It returned the following documents:

```
[
  { _id: 2, name: 'xTablet', price: 899 },
  { _id: 3, name: 'SmartTablet', price: 899 },
  { _id: 5, name: 'SmartPhone', price: 599 }
]
```

2) Using the MongoDB $nin operator to match values in an array

The following example uses the `$nin` operator to select documents where the `color` array doesn't have an element that is either `"black"` or `"white"`:

```
db.products.find({
    color: {
        $nin: ["black", "white"]
    }
}, {
    name: 1,
```

```
    color: 1
})
```

The query returned the following documents:

```
[ { _id: 3, name: 'SmartTablet', color: [ 'blue' ] } ]
```

3) Using the MongoDB $nin operator with regular expressions

The following query uses the `$nin` operator to find documents where the `color` array doesn't have an element that matches `/^g+/` and `/^w+/` regular expression:

```
db.products.find({
    color: {
        $nin: [/^g+/, /^w+/]
    }
}, {
    name: 1,
    color: 1
})
```

It returned the following documents:

```
[ { _id: 3, name: 'SmartTablet', color: [ 'blue' ] } ]
```

Summary

- Use the MongoDB `$nin` operator to select documents where the value of a field is not equal to any values in an array.
- The value list can contain literal values or regular expressions.

MongoDB $and

Summary: in this tutorial, you'll about the MongoDB `$and` operator and how to use it to perform a logical AND operation.

Introduction to the MongoDB $and operator

The `$and` is a logical query operator that allows you to carry a logical **AND** operation on an array of one or more expressions.

The following shows the syntax of the `$and` operator:

```
$and :[{expression1}, {expression2},...]
```

The `$and` operator returns `true` if all expressions evaluate to `true` .

The `$and` operator stops evaluating the remaining expressions as soon as it finds an expression that evaluates to `false` . This feature is called short-circuit evaluation.

Implicit AND operator

When you use a comma-separated list of expressions, MongoDB will carry an implicit AND operation:

```
{ field: { expression1, expression2, ... }
```

MongoDB $and operator examples

We'll use the following `products` collection:

```
db.products.insertMany([
        { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate" : ISODate
```

```
        { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate" : ISODat
        { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate" : IS
        { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate" : ISODa
        { "_id" : 5, "name" : "SmartPhone", "price" : 599, "releaseDate" : ISO
        { "_id" : 6, "name" : "xWidget", "spec" : { "ram" : 64, "screen" : 9.7
])
```

1) Using MongoDB $and operator example

The following example uses the `$and` operator to select all documents in the `products` collection where:

- the value in the price field is equal to 899 **and**
- the value in the color field is either `"white"` or `"black"`

```
db.products.find({
    $and: [{
        price: 899
    }, {
        color: {
            $in: ["white", "black"]
        }
    }]
}, {
    name: 1,
    price: 1,
    color: 1
})
```

It returned the following document:

```
[
  {
    _id: 2,
    name: 'xTablet',
```

```
    price: 899,
    color: [ 'white', 'black', 'purple' ]
  }
]
```

2) Using MongoDB $and operator with the same field

The following example uses the `$and` operator to select all documents where:

- the `price` field value equals `699` and
- the `price` field value exists

```
db.products.find({
    $and: [{
        price: 699
    }, {
        price: {
            $exists: true
        }
    }]
}, {
    name: 1,
    price: 1,
    color: 1
})
```

Output:

```
[
  {
    _id: 4,
    name: 'SmartPad',
    price: 699,
    color: [ 'white', 'orange', 'gold', 'gray' ]
  }
]
```

The following example uses the implicit AND operator and returns the same result:

```
db.products.find({
    price: {
        $eq: 699,
        $exists: true
    }
}, {
    name: 1,
    price: 1,
    color: 1
})
```

Summary

- Use the MongoDB `$and` operator to perform a logical AND operation.
- MongoDB performs an implicit AND operation if you specify a comma-separated list of expressions.

MongoDB $or

Summary: in this tutorial, you'll about the MongoDB `$or` operator and how to use it to perform a logical OR operation.

Introduction to the MongoDB $or operator

The `$or` is a logical query operator that carries a logical **OR** operation on an array of one or more expressions and selects the documents that satisfy at least one expression.

Here is the syntax of the `$or` operator:

```
$or:[{expression1}, {expression2},...]
```

MongoDB $or operator examples

We'll use the following `products` collection:

```
db.products.insertMany([
        { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate" : ISODate
        { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate" : ISODat
        { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate" : IS
        { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate" : ISODa
        { "_id" : 5, "name" : "SmartPhone", "price" : 599, "releaseDate" : ISO
])
```

1) Using MongoDB $or operator example

The following example uses the `$or` operator to select all documents in the `products` collection where the value in the price field equals 799 or 899:

```
db.products.find({
    $or: [{
        price: 799
    }, {
        price: 899
    }]
}, {
    name: 1,
    price: 1
})
```

It returned the following documents:

```
[
  { _id: 1, name: 'xPhone', price: 799 },
  { _id: 2, name: 'xTablet', price: 899 },
  { _id: 3, name: 'SmartTablet', price: 899 }
]
```

Since this example checks equality for the same price field, you should use the $in operator instead:

```
db.products.find({
    price: {
        $in: [799, 899]
    }
}, {
    name: 1,
    price: 1
})
```

2) Using MongoDB $or operator to select documents where the value of a field is in a range

The following example uses the `$or` operator to select all documents where the price is less than 699 or greater than 799:

```
db.products.find({
    $or: [
        { price: {$lt: 699} },
        { price: {$gt: 799} }
    ]
}, {
    name: 1,
    price: 1
})
```

Output:

```
[
  { _id: 2, name: 'xTablet', price: 899 },
  { _id: 3, name: 'SmartTablet', price: 899 },
  { _id: 5, name: 'SmartPhone', price: 599 }
]
```

Summary

- Use the MongoDB `$or` operator to perform a logical OR operation on a list of expressions and select documents that satisfy at least one expression.

MongoDB $not

Summary: in this tutorial, you'll learn how to use the MongoDB $not operator to perform a logical NOT operator.

Introduction to the MongoDB $not operator

The following shows the syntax of the `$not` operator:

```
{ field: { $not: { <expression> } } }
```

The `$not` operator is a logical query operator that performs a logical NOT operation on a specified `<expression>` and selects documents that do not match the `<expression>`. This includes the documents that do not contain the `field`.

MongoDB $not operator examples

We'll use the following `products` collection:

```
db.products.insertMany([
        { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate" : ISODate
        { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate" : ISODat
        { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate" : IS
        { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate" : ISODa
        { "_id" : 5, "name" : "SmartPhone", "price" : 599, "releaseDate" : ISO
        { "_id" : 6, "name" : "xWidget", "spec" : { "ram" : 64, "screen" : 9.7
])
```

1) Using MongoDB $not operator to select documents

The following example shows how to use the `$not` operator to find (https://www.mongodbtutorial.org/mongodb-crud/mongodb-find/) documents where:

- the `price` field is not greater than `699`.
- do not contain the `price` field.

```
db.products.find({
    price: {
        $not: {
            $gt: 699
        }
    }
}, {
    name: 1,
    price: 1
})
```

It returned the following documents:

```
[
  { _id: 4, name: 'SmartPad', price: 699 },
  { _id: 5, name: 'SmartPhone', price: 599 },
  { _id: 6, name: 'xWidget' }
]
```

Notice that the `{ $not: { $gt: 699 } }` is different from the `$lte` (https://www.mongodbtutorial.org/mongodb-crud/mongodb-lte/) operator. The `{ $lte : 699 }` returns documents where the `price` field exists and its value is less than or equal to `699`.

The following example uses the `$lte` operator:

```
db.products.find({
    price: {
        $lte: 699
    }
}, {
```

```
    name: 1,
    price: 1
})
```

It returned the following documents:

```
[
  { _id: 4, name: 'SmartPad', price: 699 },
  { _id: 5, name: 'SmartPhone', price: 599 }
]
```

As you can see clearly from the output, the result of the query that uses the `$lte` operator does not include the documents where the `price` field does not exist.

2) Using MongoDB $not operator to select documents based on expressions

The following example uses the `$not` operator to select documents from the `products` collection where the value of the field does not match the regular expression `/^smart+/i` :

```
db.products.find({
    name: {
        $not: /^Smart+/
    }
}, {
    name: 1
})
```

The regular expression `/^Smart+/` matches any string that starts with the string `smart` and is followed by any number of characters.

The query returns the following documents:

```
[
  { _id: 1, name: 'xPhone' },
  { _id: 2, name: 'xTablet' },
```

```
    { _id: 6, name: 'xWidget' }
]
```

Summary

- Use the MongoDB `$not` operator to perform a logical NOT operation on a specified `<expression>` and selects documents that do not match the `<expression>` .

MongoDB $nor

Summary: in this tutorial, you'll learn how to use the MongoDB `$nor` operator to perform a logical NOR operator.

Introduction to the MongoDB $nor operator

The `$nor` is a logical query operator that allows you to perform a logical NOR operation on a list of one or more query expressions and selects documents that fail all the query expressions.

The syntax of the `$nor` operator is as follows:

```
{ $nor: [ { <expression1> }, { <expression2> },...] }
```

MongoDB $nor operator examples

We'll use the following `products` collection:

```
db.products.insertMany([
        { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate" : ISODate
        { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate" : ISODat
        { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate" : IS
        { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate" : ISODa
        { "_id" : 5, "name" : "SmartPhone", "price" : 599, "releaseDate" : ISO
        { "_id" : 6, "name" : "xWidget", "spec" : { "ram" : 64, "screen" : 9.7
])
```

The following example uses the `$nor` operator to select documents from the `products` collection:

```
db.products.find({
    $nor :[
```

```
            { price: 899},
            { color: "gold"}
        ]
    }, {
        name: 1,
        price: 1,
        color: 1
    })
```

It returns documents where:

- the value is the `price` field is not `899`
- and the `color` array does not have any `"gold"` element.

including the documents that do not contains these fields.

It returned the followig documents:

```
{ "_id" : 1, "name" : "xPhone", "price" : 799, "color" : [ "white", "black" ]
{ "_id" : 6, "name" : "xWidget", "color" : [ "black" ] }
```

Summary

- Use the MongodB `$nor` operator to perform a logical NOR operation on a list of query expressions and select documents that fail all the query expressions.

MongoDB $exists

Summary: in this tutorial, you'll learn how to use the MongoDB `$exists` operator.

Introduction to the MongoDB $exists operator

The `$exists` is an element query operator that has the following syntax:

```
{ field: { $exists: <boolean_value> } }
```

When the `<boolean_value>` is true, the `$exists` operator matches the documents that contain the `field` with any value including `null` .

If the `<boolean_value>` is `false` , the `$exists` operator matches the documents that don't contain the **field**.

The MongoDB `$exists` doesn't correspond to the `EXISTS (https://www.sqltutorial.org/sql-exists/)` operator in SQL.

> Notice that MongoDB 4.2 or later doesn't treat the `$type: 0` as the synonym for `$exists:false` anymore.

MongoDB $exists operator examples

We'll use the following products collection:

```
db.products.insertMany([
        { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate" : ISODate
        { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate" : ISODat
        { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate" : IS
        { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate" : ISODa
```

```
        { "_id" : 5, "name" : "SmartPhone", "price" : 599, "releaseDate" : ISO
        { "_id" : 6, "name" : "xWidget", "spec" : { "ram" : 64, "screen" : 9.7
        { "_id" : 7, "name" : "xReader","price": null, "spec" : { "ram" : 64,
])
```

1) Using the MongoDB $exists operator example

The following example uses the `$exists` operator to select documents where the `price` field exists:

```
db.products.find(
   {
 price: {
 $exists: true
 }
},
   {
 name: 1,
 price: 1
 }
)
```

It returned the following documents:

```
{ "_id" : 1, "name" : "xPhone", "price" : 799 }
{ "_id" : 2, "name" : "xTablet", "price" : 899 }
{ "_id" : 3, "name" : "SmartTablet", "price" : 899 }
{ "_id" : 4, "name" : "SmartPad", "price" : 699 }
{ "_id" : 5, "name" : "SmartPhone", "price" : 599 }
{ "_id" : 7, "name" : "xReader", "price" : null }
```

In this example, the `$exists` operator matches the documents that have the `price` field including the non-null and null values.

The following query uses the `$exists` operator that select documents whose `price` field exists and has a value greater than `799` :

```
db.products.find({
    price: {
        $exists: true,
        $gt: 699
    }
}, {
    name: 1,
    price: 1
});
```

Output:

```
{ "_id" : 1, "name" : "xPhone", "price" : 799 }
{ "_id" : 2, "name" : "xTablet", "price" : 899 }
{ "_id" : 3, "name" : "SmartTablet", "price" : 899 }
```

2) Using the MongoDB $exists operator to query documents that don't have a specified field

The following example uses the `$exists` operator to select documents that **don't** have the `price` field:

```
db.products.find({
    price: {
        $exists: false
    }
}, {
    name: 1,
    price: 1
});
```

It returned one document that doesn't have the `price` field:

```
{ "_id" : 6, "name" : "xWidget" }
```

Summary

- Use the `{ field: {$exists: true} }` to select documents that contain the `field`. It also includes the documents where the `field` contains `null`.
- Use the `{ field: {$exists: false} }` to match documents where the `field` doesn't exist.

MongoDB $type

Summary: in this tutorial, you'll how to use the MongoDB `$type` operator to select documents where the value of a field is an instance of a BSON type.

Introduction to the MongoDB $type operator

Sometimes, you need to deal with highly unstructured data where **data types are unpredictable**. In this case, you need to use the `$type` operator.

The `$type` is an **element query operator** that allows you to select documents where the value of a field is an instance of a specified BSON type.

The `$type` operator has the following syntax:

```
{ field: { $type: <BSON type> } }
```

The `$type` operator also accepts a list of BSON types like this:

```
{ field: { $type: [ <BSON type1> , <BSON type2>, ... ] } }
```

In this syntax, the `$type` operator selects the documents where the type of the `field` matches any BSON type on the list.

MongoDB provides you with three ways to identify a BSON type: string, number, and alias. The following table lists the BSON types identified by these three forms:

Type	Number	Alias
Double	1	"double"
String	2	"string"

Type	Number	Alias
Object	3	“object”
Array	4	“array”
Binary data	5	“binData”
ObjectId	7	“objectId”
Boolean	8	“bool”
Date	9	“date”
Null	10	“null”
Regular Expression	11	“regex”
JavaScript	13	“javascript”
32-bit integer	16	“int”
Timestamp	17	“timestamp”
64-bit integer	18	“long”
Decimal128	19	“decimal”
Min key	-1	“minKey”
Max key	127	“maxKey”

The `$type` operator also supports the `number` alias that matches against the following BSON types:

- `double`
- `32-bit integer`
- `64-bit integer`
- `decimal`

MongoDB $type operator examples

We'll use the following `products` collection:

```
db.products.insertMany([
        { "_id" : 1, "name" : "xPhone", "price" : "799", "releaseDate" : ISODa
        { "_id" : 2, "name" : "xTablet", "price" : NumberInt(899), "releaseDat
        { "_id" : 3, "name" : "SmartTablet", "price" : NumberLong(899), "relea
        { "_id" : 4, "name" : "SmartPad", "price" : [599, 699, 799], "releaseD
        { "_id" : 5, "name" : "SmartPhone", "price" : ["599",699], "releaseDat
        { "_id" : 6, "name" : "xWidget", "spec" : { "ram" : 64, "screen" : 9.7
])
```

This `products` collection contains the `price` field that has int, double, long values.

1) Using the $type operator example

The following example uses the `$type` operator to query documents from the `products` collection where the `price` field is the `string` type or is an array containing an element that is a `string` type.

```
db.products.find({
    price: {
        $type: "string"
    }
}, {
    name: 1,
    price: 1
})
```

It returned the following documents:

```
{ "_id" : 1, "name" : "xPhone", "price" : "799" }
{ "_id" : 5, "name" : "SmartPhone", "price" : [ "599", 699 ] }
```

Since the `string` type corresponds to the number 2 (see the BSON types table above), you can use the number 2 in the query instead:

```
db.products.find({
    price: {
        $type: 2
    }
}, {
    name: 1,
    price: 1
})
```

2) Using the $type operator with the number alias example

The following example uses the `$type` operator with the `number` alias to select documents where the value of the `price` field is the BSON type `int` , `long` , or `double` or is an array that contains a number:

```
db.products.find({
    price: {
        $type: "number"
    }
}, {
    name: 1,
    price: 1
})
```

It returned the following documents:

```
{ "_id" : 2, "name" : "xTablet", "price" : 899 }
{ "_id" : 3, "name" : "SmartTablet", "price" : NumberLong(899) }
{ "_id" : 4, "name" : "SmartPad", "price" : [ 599, 699, 799 ] }
{ "_id" : 5, "name" : "SmartPhone", "price" : [ "599", 699 ] }
```

3) Using the $type operator to query documents with array type example

The following query use the `$type` operator to select the documents in which the `price` field is an array:

```
db.products.find({
    price: {
        $type: "array"
    }
}, {
    name: 1,
    price: 1
})
```

It returned the following documents:

```
{ "_id" : 4, "name" : "SmartPad", "price" : [ 599, 699, 799 ] }
{ "_id" : 5, "name" : "SmartPhone", "price" : [ "599", 699 ] }
```

4) Using the $type operator to query documents with multiple types

The following query uses the `$type` operator to select documents where the `price` field is either number or string or an array that has an element is number or string:

```
db.products.find({
    price: {
        $type: ["number", "string"]
    }
}, {
    name: 1,
    price: 1
})
```

It matched the following documents:

```
{ "_id" : 1, "name" : "xPhone", "price" : "799" }
{ "_id" : 2, "name" : "xTablet", "price" : 899 }
```

```
{ "_id" : 3, "name" : "SmartTablet", "price" : NumberLong(899) }
{ "_id" : 4, "name" : "SmartPad", "price" : [ 599, 699, 799 ] }
{ "_id" : 5, "name" : "SmartPhone", "price" : [ "599", 699 ] }
```

Notice that the result doesn't include the document with `_id 6` because this document doesn't have the `price` field.

Summary

- Use the `{ field: { $type: <BSON type> } }` to select the documents where the value of a field is an instance of a specified BSON type.
- Use the `{ field: { $type: [ <BSON type1> , <BSON type2>, ... ] } }` to select documents where the value of the field matches against one of the BSON types on the list.

MongoDB $size

Summary: in this tutorial, you'll how to use the MongoDB `$size` operator to match documents that have an array containing a specified number of elements.

Introduction to MongoDB $size operator

The `$size` is an array query operator that allows you to select documents that have an array containing a specified number of elements.

The `$size` operator has the following syntax:

```
{ array_field: {$size: element_count} }
```

In this syntax, you specify the `element_count` after the `$size` operator to match all documents where the `array_field` has exact `element_count` elements.

MongoDB $size operator examples

We'll use the following products collection:

```
db.products.insertMany([
        { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate" : ISODate
        { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate" : ISODat
        { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate" : IS
        { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate" : ISODa
        { "_id" : 5, "name" : "SmartPhone", "price" : 599, "releaseDate" : ISO
        { "_id" : 6, "name" : "xWidget", "spec" : { "ram" : 64, "screen" : 9.7
])
```

1) Using MongoDB $size operator to select documents that have an array containing a number of elements

The following example uses the `$size` operator to select documents whose `color` array has two elements:

```
db.products.find({
    color: {
        $size: 2
    }
}, {
    name: 1,
    color: 1
})
```

It returned the following document:

```
{ "_id" : 1, "color" : [ "white", "black" ], "name" : "xPhone" }
```

2) Using MongoDB $size operator with the $or operator example

The following example shows how to use `$size` operator with the `$or` (https://www.mongodbtutorial.org/mongodb-crud/mongodb-or/) operator to select documents whose the `color` array has one or two elements:

```
db.products.find({
    $or: [{
            color: {
                $size: 1
            }
        },
        {
            color: {
                $size: 2
            }
```

```
            }
        ]
    }, {
        name: 1,
        color: 1
    })
```

It returned the following documents:

```
{ "_id" : 1, "color" : [ "white", "black" ], "name" : "xPhone" }
{ "_id" : 3, "color" : [ "blue" ], "name" : "SmartTablet" }
```

Summary

- Use the `$size` operator to select documents that contains an array with a specified number of elements.

MongoDB $all

Summary: in this tutorial, you'll learn how to use the MongoDB `$all` operator to select documents.

Introduction to the MongoDB $all operator

The `$all` is an array query operator that allows you to find (https://www.mongodbtutorial.org/mongodb-crud/mongodb-find/) the documents where the value of a field is an array that contains all the specified elements.

The `$all` operator has the following syntax:

```
{ <arrayField>: { $all: [element1, element2, ...]} }
```

If the array followed the `$all` operator is empty, then the `$all` operator matches no documents.

When the array followed the `$all` operator contains a single element, you should use the contain expression instead:

```
{ <arrayField>: element1 }
```

$all and $and

The following expression that uses the `$all` operator:

```
{ arrayField: {$all: [element1, element2]} }
```

is equivalent to the following expression that use the `$and` (https://www.mongodbtutorial.org/mongodb-crud/mongodb-and/) operator:

```
{ $and: [{ arrayField: element1}, {arrayField: element2} ]}
```

MongoDB $all operator examples

We'll use the following `products` collection:

```
db.products.insertMany([
        { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate" : ISODate
        { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate" : ISODat
        { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate" : IS
        { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate" : ISODa
        { "_id" : 5, "name" : "SmartPhone", "price" : 599, "releaseDate" : ISO
])
```

1) Using MongoDB $all operator to match values

The following example uses the `$all` operator to query the `products` collection for documents where the value of the `color` field is an array that includes `"black"` and `"white"` :

```
db.products.find({
    color: {
        $all: ["black", "white"]
    }
}, {
    name: 1,
    color: 1
})
```

It returned the following documents. Notice that the order of elements in the array is not important.

```
{ "_id" : 1, "name" : "xPhone", "color" : [ "white", "black" ] }
{ "_id" : 2, "name" : "xTablet", "color" : [ "white", "black", "purple" ] }
```

Functionally speaking, the above query is equivalent to the following query that uses the `$and` operator:

```
db.products.find({
    $and: [
        {color: "black"},
        {color: "white"}
    ]
}, {
    name: 1,
    color: 1
})
```

Summary

- Use the `$all` operator to select the documents where the value of a field is an array that contains all the specified elements.

MongoDB $elemMatch

Summary: in this tutorial, you'll learn how to use the MongoDB `$elemMatch` operator to select documents from a collection.

Introduction to the MongoDB $elemMatch operator

The `$elemMatch` is an array query operator that matches documents that contain an array field and the array field has at least one element that satisfies all the specified queries.

The `$elemMatch` has the following syntax:

```
{ <arrayField>: {$elemMatch: { <query1>, <query2>, ...} } }
```

In this syntax:

- First, specify the name of the array field.
- Second, specify a list of queries that you want at least one element in the `<arrayField>` to meet the query criteria.

> Notice that you cannot specify a `$where` expression or a `$text` query expression in an `$elemMatch` .

MongoDB $elemMatch operator examples

We'll use the following `products` collection for the demonstration:

```
db.products.insertMany([
        { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate" : ISODate
        { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate" : ISODat
        { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate" : IS
```

```
        { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate" : ISODa
        { "_id" : 5, "name" : "SmartPhone", "price" : 599, "releaseDate" : ISO
])
```

1) Using the MongDB $elemMatch opeator example

The following example uses the `$elemMatch` operator to query documents from the `products` collection:

```
db.products.find({
    storage: {
        $elemMatch: {
            $lt: 128
        }
    }
}, {
    name: 1,
    storage: 1
});
```

It matches the documents where the `storage` is an array that contains at least one element less than 128:

```
[
  { _id: 1, name: 'xPhone', storage: [ 64, 128, 256 ] },
  { _id: 3, name: 'SmartTablet', storage: [ 16, 64, 128 ] }
]
```

Summary

- Use the `$elemMatch` operator to select documents that have an array field. And the array field has at least one element that satisfies specified query criteria.

MongoDB sort

Summary: in this tutorial, you'll learn how to use the MongoDB `sort()` method to sort the matching documents returned by a query in ascending or descending order.

Introduction to MongoDB sort() method

To specify the order of the documents returned by a query, you use the `sort()` method:

```
cursor.sort({field1: order, field2: order, ...})
```

The `sort()` method allows you to sort the matching documents by one or more fields (`field1` , `field2` , ...) in ascending or descending order.

The `order` takes two values: `1` and `-1` . If you specify `{ field: 1 }` , the `sort()` will sort the matching documents by the `field` in ascending order:

```
cursor.sort({ field: 1 })
```

If you specify `{ field: -1}` , the `sort()` method will sort matching documents by the `field` in descending order:

```
cursor.sort({field: -1})
```

The following sorts the returned documents by the `field1` in ascending order and `field2` in descending order:

```
cursor.sort({field1: 1, field2: -1});
```

It's straightforward to compare values of the same type. However, it is not the case for comparing the values of different BSON types.

MongoDB uses the following comparison order from lowest to highest for comparing values of different BSON types:

1. MinKey (internal type)
2. Null
3. Numbers (ints, longs, doubles, decimals)
4. Symbol, String
5. Object
6. Array
7. BinData
8. ObjectId
9. Boolean
10. Date
11. Timestamp
12. Regular Expression
13. MaxKey (internal type)

For more information on comparison/sort order, check out this page (https://docs.mongodb.com/manual/reference/bson-type-comparison-order/#bson-types-comparison-order) .

MongoDB sort() method examples

We'll use the following `products` collection to illustrate how the `sort()` method works.

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate" : ISODate("20
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate" : ISODate("2
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate" : ISODat
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate" : ISODate("
    { "_id" : 5, "name" : "SmartPhone", "price" : 599, "releaseDate" : ISODate
```

```
    { "_id" : 6, "name" : "xWidget", "spec" : { "ram" : 64, "screen" : 9.7, "c
    { "_id" : 7, "name" : "xReader", "price" : null, "spec" : { "ram" : 64, "s
])
```

1) Sorting document by one field examples

The following query returns all documents from the `products` collection where the `price` field exists (https://www.mongodbtutorial.org/mongodb-crud/mongodb-exists/) . For each document, it selects the `_id` , `name` , and `price` fields:

```
db.products.find({
    'price': {
        $exists: 1
    }
}, {
    name: 1,
    price: 1
})
```

Output:

```
[
  { _id: 1, name: 'xPhone', price: 799 },
  { _id: 2, name: 'xTablet', price: 899 },
  { _id: 3, name: 'SmartTablet', price: 899 },
  { _id: 4, name: 'SmartPad', price: 699 },
  { _id: 5, name: 'SmartPhone', price: 599 },
  { _id: 7, name: 'xReader', price: null }
]
```

To sort the products by prices in ascending order, you use the `sort()` method like this:

```
db.products.find({
    'price': {
```

```
        $exists: 1
    }
}, {
    name: 1,
    price: 1
}).sort({
    price: 1
})
```

Output:

```
[
  { _id: 7, name: 'xReader', price: null },
  { _id: 5, name: 'SmartPhone', price: 599 },
  { _id: 4, name: 'SmartPad', price: 699 },
  { _id: 1, name: 'xPhone', price: 799 },
  { _id: 2, name: 'xTablet', price: 899 },
  { _id: 3, name: 'SmartTablet', price: 899 }
]
```

In this example, the `sort()` method places the document whose price is `null` first, and then the documents with the prices from lowest to highest.

To sort the documents in descending order, you change the value of the price field to `-1` as shown in the following query:

```
db.products.find({
    'price': {
        $exists: 1
    }
}, {
    name: 1,
    price: 1
}).sort({
    price: -1
})
```

Output:

```
[
  { _id: 2, name: 'xTablet', price: 899 },
  { _id: 3, name: 'SmartTablet', price: 899 },
  { _id: 1, name: 'xPhone', price: 799 },
  { _id: 4, name: 'SmartPad', price: 699 },
  { _id: 5, name: 'SmartPhone', price: 599 },
  { _id: 7, name: 'xReader', price: null }
]
```

In this example, the `sort()` method places the document with the highest price first and the one whose price is `null` last. (See the sort order above)

2) Sorting document by two or more fields example

The following example uses the `sort()` method to sort the products by name and price in ascending order. It selects only documents where the `price` field exists and includes the `_id`, `name`, and `price` fields in the matching documents.

```
db.products.find({
    'price': {
        $exists: 1
    }
}, {
    name: 1,
    price: 1
}).sort({
    price: 1,
    name: 1
});
```

Output:

```
[
  { _id: 7, name: 'xReader', price: null },
  { _id: 5, name: 'SmartPhone', price: 599 },
  { _id: 4, name: 'SmartPad', price: 699 },
  { _id: 1, name: 'xPhone', price: 799 },
  { _id: 3, name: 'SmartTablet', price: 899 },
  { _id: 2, name: 'xTablet', price: 899 }
]
```

In this example, the `sort()` method sorts the products by prices first. Then it sorts the sorted result set by names.

If you look at the result set more closely, you'll see that the products with `_id` 3 and 2 have the same price `899`. The `sort()` method places the `SmartTablet` before the `xTablet` based on the ascending order specified by the `name` field.

The following example sorts the products by prices in ascending order and sorts the sorted products by names in descending order:

```
db.products.find({
    'price': {
        $exists: 1
    }
}, {
    name: 1,
    price: 1
}).sort({
    price: 1,
    name: -1
})
```

Output:

```
[
  { _id: 7, name: 'xReader', price: null },
  { _id: 5, name: 'SmartPhone', price: 599 },
```

```
    { _id: 4, name: 'SmartPad', price: 699 },
    { _id: 1, name: 'xPhone', price: 799 },
    { _id: 2, name: 'xTablet', price: 899 },
    { _id: 3, name: 'SmartTablet', price: 899 }
]
```

In this example, the `sort()` method sorts the products by prices in ascending order. However, it sorts sorted products by names in descending order.

Unlike the previous example, the `sort()` places the `xTable` before `SmartTablet` .

3) Sorting documetns by dates

The following example sorts the documents from the `products` collection by values in the `releaseDate` field. It selects only document whose `releaseDate` field exists and includes the `_id` , `name` , and `releaseDate` fields in the matching documents:

```
db.products.find({
    releaseDate: {
        $exists: 1
    }

}, {
    name: 1,
    releaseDate: 1
}).sort({
    releaseDate: 1
});
```

Output:

```
[
  {
    _id: 1,
    name: 'xPhone',
    releaseDate: ISODate("2011-05-14T00:00:00.000Z")
```

```
    },
    {
      _id: 2,
      name: 'xTablet',
      releaseDate: ISODate("2011-09-01T00:00:00.000Z")
    },
    {
      _id: 3,
      name: 'SmartTablet',
      releaseDate: ISODate("2015-01-14T00:00:00.000Z")
    },
    {
      _id: 4,
      name: 'SmartPad',
      releaseDate: ISODate("2020-05-14T00:00:00.000Z")
    },
    {
      _id: 5,
      name: 'SmartPhone',
      releaseDate: ISODate("2022-09-14T00:00:00.000Z")
    }
  ]
```

In this example, the `sort()` method places the documents with the `releaseDate` in ascending order.

The following query sorts the products by the values in the `releaseDate` field in descending order:

```
db.products.find({
    releaseDate: {
        $exists: 1
    }

}, {
    name: 1,
```

```
    releaseDate: 1
}).sort({
    releaseDate: -1
});
```

Output:

```
[
  {
    _id: 5,
    name: 'SmartPhone',
    releaseDate: ISODate("2022-09-14T00:00:00.000Z")
  },
  {
    _id: 4,
    name: 'SmartPad',
    releaseDate: ISODate("2020-05-14T00:00:00.000Z")
  },
  {
    _id: 3,
    name: 'SmartTablet',
    releaseDate: ISODate("2015-01-14T00:00:00.000Z")
  },
  {
    _id: 2,
    name: 'xTablet',
    releaseDate: ISODate("2011-09-01T00:00:00.000Z")
  },
  {
    _id: 1,
    name: 'xPhone',
    releaseDate: ISODate("2011-05-14T00:00:00.000Z")
  }
]
```

4) Sorting documents by fields in embedded documents

The following example sorts the products by the values in the `ram` field in the `spec` embedded documents. It includes the `_id`, `name`, and `spec` fields in the matching documents.

```
db.products.find({}, {
    name: 1,
    spec: 1
}).sort({
    "spec.ram": 1
});
```

Output:

```
[
  { _id: 1, name: 'xPhone', spec: { ram: 4, screen: 6.5, cpu: 2.66 } },
  {
    _id: 5,
    name: 'SmartPhone',
    spec: { ram: 4, screen: 9.7, cpu: 1.66 }
  },
  {
    _id: 4,
    name: 'SmartPad',
    spec: { ram: 8, screen: 9.7, cpu: 1.66 }
  },
  {
    _id: 3,
    name: 'SmartTablet',
    spec: { ram: 12, screen: 9.7, cpu: 3.66 }
  },
  {
    _id: 2,
    name: 'xTablet',
    spec: { ram: 16, screen: 9.5, cpu: 3.66 }
  },
```

```
    {
      _id: 6,
      name: 'xWidget',
      spec: { ram: 64, screen: 9.7, cpu: 3.66 }
    },
    {
      _id: 7,
      name: 'xReader',
      spec: { ram: 64, screen: 6.7, cpu: 3.66 }
    }
]
```

Summary

- Use the `sort()` method to sort the documents by one or more fields.
- Specify `{ field: 1 }` to sort documents by the `field` in ascending order and `{ field: -1 }` to sort documents by the `field` in descending order.
- Use the dot notation `{ "embeddedDoc.field" : 1 }` to sort the documents by the `field` in the embedded documents (`embeddedDoc`).

MongoDB limit

Summary: in this tutorial, you'll learn how to use the MongoDB `limit()` method to specify the number of documents returned by a query.

Introduction to MongoDB limit() method

The `find()` (https://www.mongodbtutorial.org/mongodb-crud/mongodb-find/) method may return a lot of documents to the application. Typically, the application may not need that many documents.

To limit the number of returned documents, you use the `limit()` method:

```
db.collection.find(<query>).limit(<documentCount>)
```

The `<documentCount>` is in the range of -2^{31} and 2^{31}. If you specify a value for the `<documentCount>` that is out of this range, the behavior of the `limit()` is unpredictable.

If the `<documentCount>` is negative, the `limit()` returns the same number of documents as if the `<documentCount>` is positive. In addition, it closes the cursor after returning a single batch of documents.

If the result set does not fit into a single batch, the number of returned documents will be less than the specified limit.

If the `<documentCount>` is zero, then is equivalent to setting no limit.

> Note that the `limit()` is analogous to the `LIMIT` clause in SQL (https://www.sqltutorial.org/sql-limit/) .

To get the predictable result set using the `limit()` , you need to sort (https://www.mongodbtutorial.org/mongodb-crud/mongodb-sort/) the result set first before applying the method like this:

```
cursor
    .sort({...})
    .limit(<documentCount>)
```

In practice, you often use the `limit()` with the `skip()` method to paginate a collection.

The `skip()` method specifies from where the query should start returning the documents:

```
cursor.skip(<offset>)
```

The following shows the documents on the page `pageNo` with the `documentCount` documents per page:

```
db.collection.find({...}
).sort({...}
).skip(pageNo > 0 ? ( ( pageNo - 1 ) * documentCount) : 0
).limit(documentCount);
```

The `skip(<offset>)` requires the MongoDB server to scan from the beginning of the result set before starting to return the documents. When the `<offset>` increases, the `skip()` will become slower.

> Note that the `limit()` and `skip()` is analogous to the `LIMIT OFFSET` clause in SQL (https://www.sqltutorial.org/sql-limit/) .

MongoDB limit() examples

We'll use the following `products` collection:

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate" : ISODate("20
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate" : ISODate("2
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate" : ISODat
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate" : ISODate("
```

```
    { "_id" : 5, "name" : "SmartPhone", "price" : 599, "releaseDate" : ISODate
    { "_id" : 6, "name" : "xWidget", "spec" : { "ram" : 64, "screen" : 9.7, "c
    { "_id" : 7, "name" : "xReader", "price" : null, "spec" : { "ram" : 64, "s
])
```

1) Using MongoDB limit() to get the most expensive product

The following example uses the `limit()` method to get the most expensive product in the `products` collection. It includes the `_id`, `name`, and `price` fields in the returned documents:

```
db.products.find({}, {
    name: 1,
    price: 1
}).sort({
    price: -1
}).limit(1);
```

Output:

```
[ { _id: 2, name: 'xTablet', price: 899 } ]
```

In this example, we sort the products by prices in descending order and use `limit()` to select the first one.

The `products` collection has two products at the same price `899`. The returned document depends on the order of documents stored on the disk.

To get the predictable result, the sort should be unique. For example:

```
db.products.find({}, {
    name: 1,
    price: 1
}).sort({
    price: -1,
```

```
    name: 1
}).limit(1);
```

Output:

```
[ { _id: 3, name: 'SmartTablet', price: 899 } ]
```

This example sorts the document by prices in descending order. And then it sorts the sorted result set by names in ascending order. The `limit()` returns the first document in the final result set.

2) Using MongoDB limit() and skip() to get the paginated result

Suppose you want to divide the products collection into pages, each has 2 products.

The following query uses the `skip()` and `limit()` to get the documents on the second page:

```
db.products.find({}, {
    name: 1,
    price: 1
}).sort({
    price: -1,
    name: 1
}).skip(2).limit(2);
```

Output:

```
[
  { _id: 1, name: 'xPhone', price: 799 },
  { _id: 4, name: 'SmartPad', price: 699 }
]
```

Summary

- Use `limit()` to specify the number of returned documents for a query.
- Use `sort()` and `limit()` to select top / bottom N documents.

- Use `skip()` and `limit()` to paginate the documents in a collection.

MongoDB updateOne

Summary: in this tutorial, you'll learn how to use the MongoDB `updateOne()` method to update the first document in a collection that matches a condition.

Introduction to MongoDB updateOne() method

The `updateOne()` method allows you to update a single document that satisfies a condition.

The following shows the syntax of the `updateOne()` method:

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

In this syntax:

- The `filter` is a document that specifies the criteria for the update. If the `filter` matches multiple documents, then the `updateOne()` method updates only the first document. If you pass an empty document `{}` into the method, it will update the first document returned in the collection.
- The `update` is a document that specifies the change to apply.
- The `options` argument provides some options for updates that won't be covered in this tutorial.

The `updateOne()` method returns a document that contains some fields. The notable ones are:

- The `matchedCount` returns the number of matched documents.
- The `modifiedCount` returns the number of updated documents. In the case of the `updateOne()` method, it can be either 0 or 1.

$set operator

The `$set` operator allows you to replace the value of a field with a specified value. The `$set` operator has the following syntax:

```
{ $set: { <field1>: <value1>, <field2>: <value2>, ...}}
```

If the `field` doesn't exist, the `$set` operator will add the new field with the specified `value` to the document as long as the new field doesn't violate a type constraint.

If you specify the `field` with the dot notation e.g., `embededDoc.field` and the `field` does not exist, the `$set` will create the embedded document (`embedded`).

MongoDB updateOne() method examples

We'll use the following `products` collection:

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate": ISODate("201
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate": ISODate("20
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate": ISODate
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate": ISODate("2
    { "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate": ISODate("
 ])
```

1) Using the MongoDB updateOne() method to update a single document

The following example uses the `updateOne()` method to update the `price` of the document with `_id: 1` :

```
db.products.updateOne({
    _id: 1
}, {
    $set: {
        price: 899
    }
})
```

In this query:

- The `{ _id : 1 }` is the `filter` argument that matches the documents to update. In this example, it matches the document whose _id is 1.
- The `{ $set: { price: 899 } }` specifies the change to apply. It uses the `$set` operator to set the value of the `price` field to `899` .

The query returns the following result:

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

In this result document, the `matchedCount` indicates the number of matching documents (`1`) and the `modifiedCount` shows the number of the updated documents (`1`).

To verify the update, you can use the `findOne()` (https://www.mongodbtutorial.org/mongodb-crud/mongodb-findone/) method to retrieve the document `_id: 1` as follows:

```
db.products.findOne({ _id: 1 }, { name: 1, price: 1 })
```

It returned the following document:

```
{ _id: 1, name: 'xPhone', price: 899 }
```

As you can see clearly from the output, the `price` has been updated successfully.

2) Using the MongoDB updateOne() method to update the first matching document

The following query selects the documents from the `products` collection in which the value of the `price` field is 899:

```
db.products.find({ price: 899 }, { name: 1, price: 1 })
```

It returned the following documents:

```
[
  { _id: 1, name: 'xPhone', price: 899 },
  { _id: 2, name: 'xTablet', price: 899 },
  { _id: 3, name: 'SmartTablet', price: 899 }
]
```

The following example uses the `updateOne()` method to update the first matching document where the `price` field is `899`:

```
db.products.updateOne({ price: 899 }, { $set: { price: null } })
```

It updated one document as shown in the following result:

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

If you query the document with `_id: 1`, you'll see that its `price` field is updated:

```
db.products.find({ _id: 1}, { name: 1, price: 1 })
```

Output:

```
[ { _id: 1, name: 'xPhone', price: null } ]
```

3) Using the updateOne() method to update embedded documents

The following query uses the `find()` (https://www.mongodbtutorial.org/mongodb-crud/mongodb-find/) method to select the document with `_id: 4` :

```
db.products.find({ _id: 4 }, { name: 1, spec: 1 })
```

It returned the following document:

```
[
  {
    _id: 4,
    name: 'SmartPad',
    spec: { ram: 8, screen: 9.7, cpu: 1.66 }
  }
]
```

The following example uses the `updateOne()` method to update the values of the `ram` , `screen` , and `cpu` fields in the spec embedded document of the document `_id: 4` :

```
db.products.updateOne({
    _id: 4
}, {
    $set: {
        "spec.ram": 16,
        "spec.screen": 10.7,
        "spec.cpu": 2.66
    }
})
```

It returned the following document:

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
```

```
    upsertedCount: 0
}
```

If you query the document with `_id` 4 again, you'll see the change:

```
db.products.find({ _id: 4 }, { name: 1, spec: 1 })
```

Output:

```
[
  {
    _id: 4,
    name: 'SmartPad',
    spec: { ram: 16, screen: 10.7, cpu: 2.66 }
  }
]
```

4) Using the MongoDB updateOne() method to update array elements

The following example uses the `updateOne()` method to update the first and second elements of the `storage` array in the document with `_id 4` :

```
db.products.updateOne(
 { _id: 4},
 {
    $set: {
        "storage.0": 16,
        "storage.1": 32
    }
 }
)
```

Output:

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

If you query the document with `_id 4` from the `products` collection, you'll see that the first and second elements of the `storage` array have been updated:

```
db.products.find({ _id: 4 }, { name: 1, storage: 1 });
```

Output:

```
[ { _id: 4, name: 'SmartPad', storage: [ 16, 32, 1024 ] } ]
```

Summary

- Use the `updateOne()` method to update the first document within a collection that satisfies a condition.
- Use the `$set` operator to replace the value of a field with a specified value.

MongoDB updateMany

Summary: in this tutorial, you'll learn how to use the MongoDB `updateMay()` method to update all the documents that match a condition.

Introduction to MongoDB updateMany() method

The `updateMany()` method allows you to update all documents that satisfy a condition.

The following shows the syntax of the `updateMany()` method:

```
db.collection.updateMany(filter, update, options)
```

In this syntax:

- The `filter` is a document that specifies the condition to select the document for update. If you pass an empty document (`{}`) into the method, it'll update all the documents the collection.
- The `update` is a document that specifies the updates to apply.
- The `options` argument provides some options for updates that won't be covered in this tutorial.

The `updateMany()` method returns a document that contains multiple fields. The following are the notable ones:

- The `matchedCount` stores the number of matched documents.
- The `modifiedCount` stores the number of modified documents.

To form the update argument, you typically use the `$set` operator.

$set operator

The `$set` operator replaces the value of a field with a specified value. It has the following syntax:

```
{ $set: { <field1>: <value1>, <field2>: <value2>, ...}}
```

If the `field` doesn't exist, the `$set` operator will add the new field with the specified `value` to the document. If you specify the `field` with the dot notation e.g., `embededDoc.field` and the `field` does not exist, the `$set` operator will create the embedded document (`embedded`).

MongoDB updateMany() method examples

We'll use the following `products` collection:

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate": ISODate("201
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate": ISODate("20
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate": ISODate
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate": ISODate("2
    { "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate": ISODate("
 ])
```

1) Using the MongoDB updateMany() method to update multiple documents

The following example uses the `updateMany()` method to update the documents where the value of the price field is `899` :

```
db.products.updateMany(
    {  price: 899},
    { $set: {  price: 895 }}
)
```

In this query:

The `{ price : 899 }` is the `filter` argument that specified the documents to update.

The `{ $set: { price: 895} }` specifies the update to apply, which uses the `$set` operator to set the value of the `price` field to `895` .

The query returns the following result:

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 2,
  modifiedCount: 2,
  upsertedCount: 0
}
```

In this result document, the `matchedCount` stores the number of matching documents (2) and the `modifiedCount` stores the number of the updated documents (2).

To check the update, you can use the `find()` (https://www.mongodbtutorial.org/mongodb-crud/mongodb-find/) method to select the documents where the value of the `price` field is `895` as follows:

```
db.products.find({
    price: 895
}, {
    name: 1,
    price: 1
})
```

The query returned the following documents:

```
[
  { _id: 2, name: 'xTablet', price: 895 },
  { _id: 3, name: 'SmartTablet', price: 895 }
]
```

The `price` field values have been updated successfully.

2) Using the updateMany() method to update embedded documents

The following query uses the `find()` (https://www.mongodbtutorial.org/mongodb-crud/mongodb-find/) method to select the documents where the value in the `price` field is greater than 700:

```
db.products.find({
    price: { $gt: 700}
}, {
    name: 1,
    price: 1,
    spec: 1
})
```

The query returned the following documents:

```
[
  {
    _id: 1,
    name: 'xPhone',
    price: 799,
    spec: { ram: 4, screen: 6.5, cpu: 2.66 }
  },
  {
    _id: 2,
    name: 'xTablet',
    price: 895,
    spec: { ram: 16, screen: 9.5, cpu: 3.66 }
  },
  {
    _id: 3,
    name: 'SmartTablet',
    price: 895,
    spec: { ram: 12, screen: 9.7, cpu: 3.66 }
  }
]
```

The following example uses the `updateMany()` method to update the values of the `ram`, `screen`, and `cpu` fields in the `spec` embedded documents of these documents:

```
db.products.updateMany({
    price: { $gt: 700}
}, {
    $set: {
        "spec.ram": 32,
        "spec.screen": 9.8,
        "spec.cpu": 5.66
    }
})
```

The query returned the following document indicating that the three documents have been updated successfully:

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 3,
  modifiedCount: 3,
  upsertedCount: 0
}
```

3) Using the MongoDB updateMany() method to update array elements

The following example uses the `updateMany()` method to update the first and second elements of the `storage` array of the documents where the _id is 1, 2 and 3:

```
db.products.updateMany({
    _id: {
        $in: [1, 2, 3]
    }
}, {
    $set: {
        "storage.0": 16,
        "storage.1": 32
```

```
    }
})
```

Output:

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 3,
  modifiedCount: 3,
  upsertedCount: 0
}
```

If you query the documents whose `_id` is 1, 2, and 3 from the `products` collection, you'll see that the first and second elements of the `storage` array have been updated:

```
db.products.find(
   { _id: { $in: [1,2,3]}},
   { name: 1, storage: 1}
)
```

Output:

```
[
  { _id: 1, name: 'xPhone', storage: [ 16, 32, 256 ] },
  { _id: 2, name: 'xTablet', storage: [ 16, 32, 512 ] },
  { _id: 3, name: 'SmartTablet', storage: [ 16, 32, 128 ] }
]
```

Summary

- Use the `updateMany()` method to update all the documents within a collection that satisfy a condition.
- Use the `$set` operator to replace the value of a field with a specified value.

How to Use MongoDB $inc Operator in update() Method

Summary: in this tutorial, you'll learn how to use the `$inc` operator in the `update()` method to increment a field by a specified value.

Introduction to the MongoDB $inc operator

Sometimes, you need to increment the value of one or more fields by a specified value. In this case, you can use the `update()` method with the $inc operator.

The $inc operator has the following syntax:

```
{ $inc: {<field1>: <amount1>, <field2>: <amount2>, ...} }
```

In this syntax, the `amount` can be positive or negative. When it's a positive value, the $inc increases the `field` by `amount`. If the `amount` is a negative value, then the $inc decreases the `field` by the absolute value of the `amount`.

If the `field` doesn't exist, the `$inc` creates the `field` and sets the field to the specified `amount`.

MongoDB $inc operator examples

We'll use the following `products` collection:

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate": ISODate("201
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate": ISODate("20
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate": ISODate
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate": ISODate("2
```

```
    { "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate": ISODate("
])
```

1) Using the MongoDB $inc operator to increase the value of a field

The following example uses the $inc operator to increase the price of the document `_id 1` from the `products` collection by 50:

```
db.products.updateOne({
    _id: 1
}, {
    $inc: {
        price: 50
    }
})
```

It returned the following document indicating that one document matched and has been updated:

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

If you query the document _id 1, you'll see that the price has been increased:

```
db.products.find(
    {_id: 1},
    {name: 1, price: 1}
)
```

Output:

```
[ { _id: 1, name: 'xPhone', price: 849 } ]
```

2) Using the MongoDB $inc operator to decrease the value of a field

The following example uses the $inc operator to decrease the price by 150:

```
db.products.updateOne({
    _id: 1
}, {
    $inc: {
        price: -150
    }
})
```

Output:

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

The price of the document _id 1 has been decreased as shown in the output of the following query:

```
db.products.find(
    { _id: 1 },
    { name: 1, price: 1 }
)
```

Output:

```
[ { _id: 1, name: 'xPhone', price: 699 } ]
```

3) Using MongoDB $inc operator to update values of multiple fields

The following example uses the `$inc` operator to increase the value of the `price` field as well as the `ram` field of the `spec` embedded document:

```
db.products.updateOne({
    _id: 1
}, {
    $inc: {
        price: 50,
        "spec.ram": 4
    }
})
```

Output:

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

Note that to specify a field in an embedded document, you use the dot notation e.g., `"spec.ram"` .

This query selects the document _id 1 to verify the update:

```
db.products.find(
    {_id: 1},
    {name: 1, price: 1, "spec.ram": 1}
)
```

The values of the `price` and `ram` fields have been increased as shown in the following output:

```
[ { _id: 1, name: 'xPhone', price: 749, spec: { ram: 8 } } ]
```

Summary

- Use the `$inc` operator to increase a value of a field by a specified amount.
- Use a negative value in the $inc operator to decrease the value of the field.

How to Use the MongoDB $min Operator to Update Field Values

Summary: in this tutorial, you'll learn how to use the MongoDB `$min` operator in the `update()` method to update the values of one or more fields.

Introduction to the MongoDB $min operator

The `$min` operator is a field update operator that allows you to update the value of a field to a specified value if the specified value is less than (`<`) the current value of the field.

The `$min` operator has the following syntax:

```
{ $min: {<field1>: <value1>, ...} }
```

If the current value of a field is greater than or equal to the value that you want to update, the `$min` operator won't update the value.

If the field doesn't exist, the `$min` operator creates the field and sets its value to the specified value.

MongoDB $min operator example

We'll use the following `products` collection:

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate": ISODate("201
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate": ISODate("20
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate": ISODate
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate": ISODate("2
    { "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate": ISODate("
])
```

The following example uses the `$min` operator to update the price of the document _id 5:

```
db.products.updateOne({
    _id: 5
}, {
    $min: {
        price: 699
    }
})
```

The query found a matching document. However, it didn't update any:

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

The reason is that the new value 699 is greater than the current value 599.

The following example uses the `$min` operator to update the price of the document `_id 5` :

```
db.products.updateOne({
    _id: 5
}, {
    $min: {
        price: 499
    }
})
```

In this case, the `price` field of the document `_id 5` is updated to `499` :

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

This query verifies the update:

```
db.products.find({ _id: 5 }, { name: 1, price: 1 })
```

Output:

```
[ { _id: 5, name: 'SmartPhone', price: 499 } ]
```

Summary

- Use the `$min` operator to update the value of a field to a specified value when the specified value is less than the current field value.

How to Use the MongoDB $max Operator to Update Field Values

Summary: in this tutorial, you'll learn how to use the MongoDB `$max` operator to update field values to specified values.

Introduction to the MongoDB $max operator

The `$max` is a field update operator that allows you to update the value of a field to a specified value if the specified value is **greater than** (`>`) the current value of the field.

Here is the syntax of the `$max` operator:

```
{ $max: {<field1>: <value1>, ...} }
```

If the current value of a field is less than or equal to the value that you want to update, the `$max` operator won't update the value.

If the field doesn't exist, the `$max` operator creates the field and sets its value to the specified value.

MongoDB $max operator example

We'll use the following `products` collection:

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate": ISODate("201
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate": ISODate("20
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate": ISODate
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate": ISODate("2
    { "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate": ISODate("
])
```

The following example uses the `$max` operator to update the `price` of the document `_id 1` :

```
db.products.updateOne({
    _id: 1
}, {
    $max: {
        price: 699
    }
})
```

The query found one matching document but it didn't update any document as shown in the `modifiedCount` value:

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

The reason is that the new value that we want to update is `699` which is less than the current value of the price field `799` .

The following example uses the `$min` operator to update the `price` of the document `_id` 1 to 899:

```
db.products.updateOne({
    _id: 1
}, {
    $max: {
        price: 899
    }
})
```

In this case, the `price` field of the document `_id 1` is updated to `899`:

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

This query verifies the update:

```
db.products.find({ _id: 1 }, { name: 1, price: 1 })
```

Output:

```
[ { _id: 1, name: 'xPhone', price: 899 } ]
```

Summary

- Use the `$max` operator to update the value of a field to a specified value when the specified value is greater than the current value.

How to use MongoDB $mul operator to update documents

Summary: in this tutorial, you'll learn how to use the MongoDB `$mul` operator to **multiply** the value of a field by a number.

Introduction to MongoDB $mul operator

The `$mul` is a field update operator that allows you to **multiply** the value of a field by a specified number.

The `$mul` operator has the following syntax:

```
{ $mul: { <field1>: <number1>, <field2>: <number2>, ... } }
```

The `<field>` that you want to update must contain a numeric value. To specify a field in an embedded document or in an array, you use the dot notation e.g., < `embedded_doc>.<field>` or `<array>.<index>`

If the field doesn't exist in the document, the `$mul` operator creates the field and sets its value to zero.

MongoDB $mul operator examples

We'll use the following `products` collection:

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate": ISODate("201
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate": ISODate("20
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate": ISODate
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate": ISODate("2
```

```
    { "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate": ISODate("
])
```

1) Using the MongoDB $mul to multiply the value of a field

The following example uses the `$mul` operator to multiply the price of the document `_id 5` by `10%`:

```
db.products.updateOne({ _id: 5 }, { $mul: { price: 1.1 } })
```

The output document showed that the query matched one document and updated it:

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

The following query verifies the update:

```
db.products.find({
    _id: 5
}, {
    name: 1,
    price: 1
})
```

Output:

```
[ { _id: 5, name: 'SmartPhone', price: 658.9000000000001 } ]
```

2) Using the MongoDB $mul to multiply the values of array elements

The following query uses the `$mul` operator to double the values of the first, second, and third array elements in the `storage` array of the document `_id 1` :

```
db.products.updateOne({
    _id: 1
}, {
    $mul: {
        "storage.0": 2,
        "storage.1": 2,
        "storage.2": 2
    }
})
```

Output:

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

The following query uses the `findOne()` (https://www.mongodbtutorial.org/mongodb-crud/mongodb-findone/) method to select the document with `_id 1` to verify the update:

```
db.products.findOne({ _id: 1 }, { name: 1, storage: 1 })
```

Output:

```
{ _id: 1, name: 'xPhone', storage: [ 128, 256, 512 ] }
```

3) Using the $mul operator to multiply the values of a field in embedded documents

This example uses the `$mul` operator to double the values of the `ram` field in the `spec` embedded documents of all documents from the `products` collection:

```
db.products.updateMany({}, {
    $mul: {
        "spec.ram": 2
    }
})
```

Output:

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 5,
  modifiedCount: 5,
  upsertedCount: 0
}
```

The following query returns all documents from the `products` collection:

```
db.products.find({}, { name: 1, "spec.ram": 1 })
```

Output:

```
[
  { _id: 1, name: 'xPhone', spec: { ram: 8 } },
  { _id: 2, name: 'xTablet', spec: { ram: 32 } },
  { _id: 3, name: 'SmartTablet', spec: { ram: 24 } },
  { _id: 4, name: 'SmartPad', spec: { ram: 16 } },
  { _id: 5, name: 'SmartPhone', spec: { ram: 8 } }
]
```

Summary

- Use the MongoDB `$mul` operator to **multiply** the value of a field by a specified number.

How to use MongoDB $unset operator to remove a field from a document

Summary: in this tutorial, you'll learn how to use the MongoDB `$unset` operator to remove one or more fields from a document.

Introduction to the MongoDB $unset operator

Sometimes, you may want to remove one or more fields from a document. In order to do it, you can use the `$unset` operator.

The `$unset` is a field update operator that completely removes a particular field from a document.

The `$unset` operator has the following syntax:

```
{ $unset: {<field>: "", ... }}
```

In this syntax, you specify the field that you want to remove and its value. The field value isn't important and doesn't impact the operation. You can specify any value, the `$unset` will remove the field completely. If the `<field>` doesn't exist in the document, then `$unset` operator will do nothing. It also won't issue any warnings or errors.

To specify a field in an embedded document, you use the dot notation like this:

```
{ $unset: { "<embedded_doc>.<field>: "", ... }}
```

Note that the `$unset` operator doesn't remove array elements. Instead, it sets the array elements to null.

```
{ $unset: {"<array>.<index>": "", ...}
```

This behavior keeps the array size and element positions consistent.

MongoDB $unset operator examples

We'll use the following `products` collection:

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate": ISODate("201
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate": ISODate("20
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate": ISODate
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate": ISODate("2
    { "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate": ISODate("
 ])
```

1) Using the MongoDB $unset operator to remove a field from a document

The following example uses the `$unset` operator to remove the price field from the document `_id 1` in the `products` collection:

```
db.products.updateOne({
    _id: 1
}, {
    $unset: {
        price: ""
    }
})
```

Output:

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
```

```
  upsertedCount: 0
}
```

The `modifiedCount` indicated that one document has been modified. This query returns all documents from the `products` collection to verify the update:

```
db.products.find({}, { name: 1, price: 1 })
```

Output:

```
[
  { _id: 1, name: 'xPhone' },
  { _id: 2, name: 'xTablet', price: 899 },
  { _id: 3, name: 'SmartTablet', price: 899 },
  { _id: 4, name: 'SmartPad', price: 699 },
  { _id: 5, name: 'SmartPhone', price: 599 }
]
```

As you can see clearly from the output, the `$unset` operator has completely removed the `price` field from the document with `_id 1` .

2) Using the MongoDB $unset operator to remove a field in an embedded document

The following statement uses the `$unset` operator to remove the `ram` field from the `spec` embedded documents of all documents in the `products` collection:

```
db.products.updateMany({}, {
    $unset: {
        "spec.ram": ""
    }
})
```

Output:

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 5,
  modifiedCount: 5,
  upsertedCount: 0
}
```

The following query returns all documents from the `products` collection:

```
db.products.find({}, {
    spec: 1
})
```

Output:

```
[
  { _id: 1, spec: { screen: 6.5, cpu: 2.66 } },
  { _id: 2, spec: { screen: 9.5, cpu: 3.66 } },
  { _id: 3, spec: { screen: 9.7, cpu: 3.66 } },
  { _id: 4, spec: { screen: 9.7, cpu: 1.66 } },
  { _id: 5, spec: { screen: 5.7, cpu: 1.66 } }
]
```

As you can see, the `ram` field has been removed from `spec` embedded document in all documents.

3) Using the MongoDB $unset operator to set array elements to null

The following example uses the `$unset` operator to set the first elements of the storage arrays to `null`:

```
db.products.updateMany({}, { $unset: { "storage.0": "" } })
```

Output:

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 5,
  modifiedCount: 5,
  upsertedCount: 0
}
```

The following query selects the `storage` array from all documents in the `products` collection:

```
db.products.find({}, { "storage":1})
```

Output:

```
[
  { _id: 1, storage: [ null, 128, 256 ] },
  { _id: 2, storage: [ null, 256, 512 ] },
  { _id: 3, storage: [ null, 64, 128 ] },
  { _id: 4, storage: [ null, 256, 1024 ] },
  { _id: 5, storage: [ null, 256 ] }
]
```

In this example, the `$unset` operator sets the first elements of the `storage` arrays to `null` instead of removing them completely.

4) Using the MongoDB $unset operator to remove multiple fields from a document

The following statement uses the `$unset` operator to remove the `releaseDate` and `spec` fields from all the documents in the `products` collection:

```
db.products.updateMany({}, {
    $unset: {
        releaseDate: "",
        spec: ""
```

```
    }
})
```

Output:

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 5,
  modifiedCount: 5,
  upsertedCount: 0
}
```

The following query verifies the update:

```
db.products.find({}, {
    name: 1,
    storage: 1,
    releaseDate: 1,
    spec: 1
})
```

Output:

```
[
  { _id: 1, name: 'xPhone', storage: [ null, 128, 256 ] },
  { _id: 2, name: 'xTablet', storage: [ null, 256, 512 ] },
  { _id: 3, name: 'SmartTablet', storage: [ null, 64, 128 ] },
  { _id: 4, name: 'SmartPad', storage: [ null, 256, 1024 ] },
  { _id: 5, name: 'SmartPhone', storage: [ null, 256 ] }
]
```

As shown clearly from the output, the `releaseDate` and `spec` fields now are gone.

Summary

- Use the `$unset` operator to completely remove a field from a document.

How to Use the MongoDB $rename field operator

Summary: in this tutorial, you'll learn how to use the MongoDB `$rename` operator to rename a field in a document.

Introduction to the MongoDB $rename operator

Sometimes, you want to rename a field in a document e.g., when it is misspelled or not descriptive enough. In this case, you can use the `$rename` operator.

The `$rename` is a field update operator that allows you to rename a field in a document to the new one.

The `$rename` operator has the following syntax:

```
{ $rename: { <field_name>: <new_field_name>, ...}}
```

In this syntax, the `<new_field_name>` must be different from the `<field_name>` .

If the document has a field with the same name as the `<new_field_name>` , the `$rename` operator removes that field and renames the specified `<field_name>` to `<new_field_name>` .

In case the `<field_name>` doesn't exist in the document, the `$rename` operator does nothing. It also won't issue any warnings or errors.

The `$rename` operator can rename fields in embedded documents. In addition, it can move these fields in and out of the embedded documents.

MongoDB $rename field operator examples

We'll use the following `products` collection:

```
db.products.insertMany([
    { "_id" : 1, "nmea" : "xPhone", "price" : 799, "releaseDate": ISODate("201
```

```
    { "_id" : 2, "nmea" : "xTablet", "price" : 899, "releaseDate": ISODate("20
    { "_id" : 3, "nmea" : "SmartTablet", "price" : 899, "releaseDate": ISODate
    { "_id" : 4, "nmea" : "SmartPad", "price" : 699, "releaseDate": ISODate("2
    { "_id" : 5, "nmea" : "SmartPhone", "price" : 599,"releaseDate": ISODate("
])
```

1) Using MongoDB $rename to rename a field in a document

The following example uses the `$rename` operator to rename the misspelled field `nmea` to `name`:

```
db.products.updateMany({}, {
    $rename: {
        nmea: "name"
    }
})
```

In this example, the `$rename` operator changed the field name from `nmea` to `name` as indicated in the following returned document:

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 5,
  modifiedCount: 5,
  upsertedCount: 0
}
```

To verify the update, you can use the `find()` (https://www.mongodbtutorial.org/mongodb-crud/mongodb-find/) method to select all documents from the `products` collection:

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

Output:

```
[
  { _id: 1, name: 'xPhone' },
  { _id: 2, name: 'xTablet' },
  { _id: 3, name: 'SmartTablet' },
  { _id: 4, name: 'SmartPad' },
  { _id: 5, name: 'SmartPhone' }
]
```

2) Using MongoDB $rename operator to rename fields in embedded documents

The following example uses the `$rename` operator to change the `size` field of the `spec` embedded document to `screenSize`:

```
db.products.updateMany({}, {
    $rename: {
        "spec.screen": "spec.screenSize"
    }
})
```

It returned the following result:

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 5,
  modifiedCount: 0,
  upsertedCount: 0
}
```

This query uses the `find()` (https://www.mongodbtutorial.org/mongodb-crud/mongodb-find/) method to select all documents from the `products` collection:

```
db.products.find({}, {
    spec: 1
})
```

Here is the output:

```
[
  { _id: 1, spec: { ram: 4, cpu: 2.66, screenSize: 6.5 } },
  { _id: 2, spec: { ram: 16, cpu: 3.66, screenSize: 9.5 } },
  { _id: 3, spec: { ram: 12, cpu: 3.66, screenSize: 9.7 } },
  { _id: 4, spec: { ram: 8, cpu: 1.66, screenSize: 9.7 } },
  { _id: 5, spec: { ram: 4, cpu: 1.66, screenSize: 5.7 } }
]
```

As you can see from the output, the `screen` fields in the `spec` embedded documents have been renamed to `screenSize` .

3) Using the MongoDB $rename to move field out of the embedded document

The following example uses the `$rename` operator to move the `cpu` field out of the `spec` embedded document in the document `_id 1` :

```
db.products.updateOne({
 _id: 1
},
{
    $rename: {
        "spec.cpu": "cpu"
    }
})
```

Output:

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

The following query selects the document with `_id 1` to verify the rename:

```
db.products.find({ _id: 1})
```

Output:

```
[
  {
    _id: 1,
    price: 799,
    releaseDate: ISODate("2011-05-14T00:00:00.000Z"),
    spec: { ram: 4, screenSize: 6.5 },
    color: [ 'white', 'black' ],
    storage: [ 64, 128, 256 ],
    name: 'xPhone',
    cpu: 2.66
  }
]
```

As you can see clearly from the output, the `cpu` field becomes the top-level field.

4) Using the MongoDB $rename to rename a field to an existing field

The following example uses the `$rename` operator to rename the field `color` to `storage` in the document with `_id 2` .

However, the `storage` field already exists. Therefore, the `$rename` operator removes the `storage` field and renames the field `color` to `storage` :

```
db.products.updateOne({
    _id: 2
}, {
    $rename: {
        "color": "storage"
    }
})
```

Output:

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

Let's check the document `_id 2` :

```
db.products.find({ _id: 2 })
```

Output:

```
[
  {
    _id: 2,
    price: 899,
    releaseDate: ISODate("2011-09-01T00:00:00.000Z"),
    spec: { ram: 16, cpu: 3.66, screenSize: 9.5 },
    storage: [ 'white', 'black', 'purple' ],
    name: 'xTablet'
  }
]
```

Summary

- Use the MongoDB `$rename` operator to rename a field to a new one.

MongoDB Upsert

Summary: in this tutorial, you'll learn how to use the MongoDB upsert function.

Introduction to the MongoDB upsert

Upsert is a combination of **up**date and in**sert**. Upsert performs two functions:

- Update data if there is a matching document.
- Insert a new document in case there is no document matches the query criteria.

To perform an upsert, you use the following `updateMany()` method with the `upsert` option set to `true`:

```
document.collection.updateMany(query, update, { upsert: true} )
```

The `upsert` field in the third argument is set to `false` by default. This means that if you omit it, the method will only update the documents that match the `query`.

Notice that the `updateOne()` method also can upsert with the `{ upsert: true }`.

MongoDB upsert examples

We'll use the following `products` collection.

```
db.products.insertMany([
    { "_id" : 1, "nmea" : "xPhone", "price" : 799, "releaseDate": ISODate("201
    { "_id" : 2, "nmea" : "xTablet", "price" : 899, "releaseDate": ISODate("20
    { "_id" : 3, "nmea" : "SmartTablet", "price" : 899, "releaseDate": ISODate
    { "_id" : 4, "nmea" : "SmartPad", "price" : 699, "releaseDate": ISODate("2
    { "_id" : 5, "nmea" : "SmartPhone", "price" : 599,"releaseDate": ISODate("
])
```

The following query uses the `update()` method to update the price for the document `_id 6` :

```
db.products.updateMany(
    {_id: 6 },
    { $set: {price: 999} }
)
```

The query found no match and didn't update any document as shown in the following output:

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

If you pass the `{ upsert: true }` to the `updateMany()` method, it'll insert a new document. For example:

```
db.products.updateMany(
    { _id: 6 },
    { $set: {price: 999} },
    { upsert: true}
)
```

The query returns the following document:

```
{
  acknowledged: true,
  insertedId: 6,
  matchedCount: 0,
  modifiedCount: 0,
```

```
    upsertedCount: 1
}
```

The output indicates that there was no matching document (`matchedCount` is zero) and the `updateMany()` method didn't update any document.

However, the `updateMany()` method inserted one document and returned the `id` of the new document stored in the `upsertedId` field.

If you query the document with `_id 6` from the `products` collection, you'll see the new document with the `price` field:

```
db.products.find({_id:6})
```

Output:

```
[ { _id: 6, price: 999 } ]
```

Summary

- Use the `{ upsert: true }` argument in the `updateMany()` or `updateOne()` method to perform an upsert.

MongoDB deleteOne

Summary: in this tutorial, you'll learn how to use the MongoDB `deleteOne()` method to delete a single document from a collection.

Introduction to the MongoDB deleteOn() method

The `deleteOne()` method allows you to delete a single document from a collection.

The `deleteOne()` method has the following syntax:

```
db.collection.deleteOne(filter, option)
```

The `deleteOne()` method accepts two arguments:

- `filter` is an required argument. The `filter` is a document that specifies the condition for matching the document to remove. If you pass an empty document `{}` into the method, it'll delete the first document in the collection.
- `option` is an optional argument. The `option` is a document that specifies the deletion options.

The `deleteOne()` method returns a document containing the `deleteCount` field that stores the number of deleted documents.

To delete all documents that match a condition from a collection, you use the `deleteMany()` method.

MongoDB deleteCount() method examples

We'll use the following `products` collection:

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate": ISODate("201
```

```
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate": ISODate("20
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate": ISODate
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate": ISODate("2
    { "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate": ISODate("
])
```

1) Using the deleteOne() method to delete a single document

The following example uses the `deleteOne()` method to delete a document with the `_id` is `1` from the `products` collection:

```
db.products.deleteOne({ _id: 1 })
```

The query returned the following document:

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

The `deleteCount` field returned `1` indicating the number of deleted documents.

2) Using the deleteOne() method to delete the first document from a collection

The following query uses the `deleteOne()` method to remove the first document returned from the `products` collection:

```
db.products.deleteOne({})
```

It returned the following document:

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

In this example, we passed an empty document `{}` to the `deleteOne()` method. Therefore, it removed the first document from the `products` collection.

Summary

- Use the MongoDB `deleteOne()` method to remove a single document that matches a condition.
- Pass an empty document into the `deleteOne()` method to delete the first document from the collection.

MongoDB deleteMany

Summary: in this tutorial, you'll learn how to use the MongoDB `deleteMany()` method to delete all documents that match a condition from a collection.

Introduction to MongoDB deleteMany() method

The `deleteMany()` method allows you to remove all documents that match a condition from a collection.

The `deleteMany()` has the following syntax:

```
db.collection.deleteMany(filter, option)
```

In this syntax:

- `filter` is a document that specifies deletion criteria. If the filter is an empty document `{}`, the `deleteMany()` method will delete all documents from the collection.
- `option` is a document that specifies the deletion option.

The `deleteMany()` returns a document containing the `deleteCount` field that stores the number of deleted documents.

To delete a single document from a collection, you can use the `deleteOne()` (https://www.mongodbtutorial.org/mongodb-crud/mongodb-deleteone/) method.

MongoDB deleteMany() method examples

We'll use the following `products` collection.

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate": ISODate("201
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate": ISODate("20
```

```
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate": ISODate
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate": ISODate("2
    { "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate": ISODate("
])
```

1) Using the deleteMany() method to delete multiple documents

The following example uses the `deleteMany()` method to delete all documents where the value in the `price` field is `899` :

```
db.products.deleteMany({ price: 899 })
```

In this example, the `filter` argument is `{ price: 899 }` that matches documents whose price is `899` .

It returned the following document:

```
{ "acknowledged" : true, "deletedCount" : 2 }
```

The `deleteCount` field indicates that the number of deleted documents is 2.

2) Using the deleteMany() method to delete all documents

The following query uses the `deleteMany()` method to delete all documents from the `products` collection:

```
db.products.deleteMany({})
```

Output:

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

In this example, we passed an empty document `{}` into the `deleteMany()` method. Therefore, it deleted all documents from the `products` collection.

Summary

- Use the `deleteMany()` method to remove all documents that match a condition from a collection.
- Pass an empty document `{}` into the `deleteMany()` method to remove all documents from the collection.