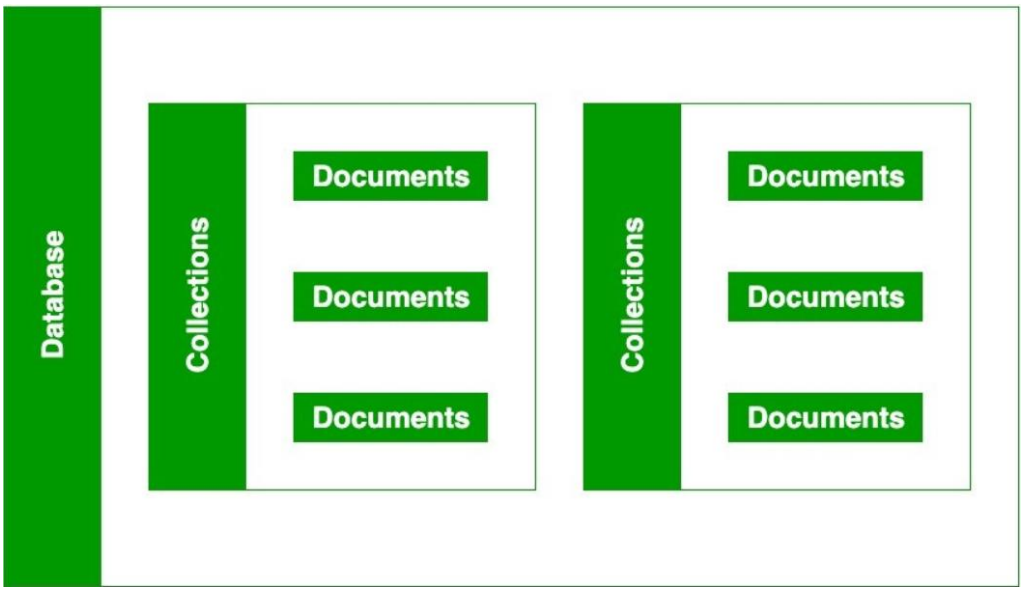
MongoDB – Database ,Collection and Document

Databases, collections, documents are important parts of MongoDB without them you are not able to store data on the MongoDB server. A Database contains a collection, and a collection contains documents and the documents contain data, they are related to each other.



DATABASE:

In MongoDB, a database is a logical container that groups related collections of documents. While it provides a way to organize your data, it doesn't directly store information itself. Think of it like a library building - the library itself doesn't hold the books, but it houses shelves (collections) that categorize and store the books (documents).

Why Databases in MongoDB?

MongoDB, like other databases, offers several advantages over storing data in plain files or code:

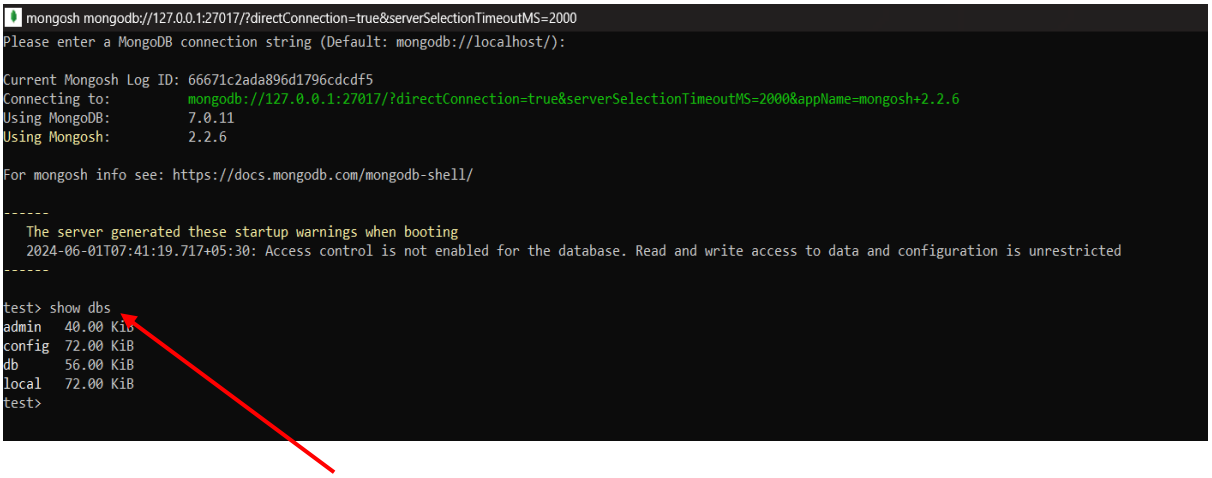
- **Organization and Structure:** Databases provide a structured way to store and manage your data. Unlike flat files, MongoDB organizes data into collections (similar to tables in relational databases) of documents. This structure makes it easier to find, retrieve, and update specific information.
- **Scalability:** Databases can handle large amounts of data efficiently. MongoDB allows you to scale horizontally by adding more servers to your cluster, distributing the load and enabling you to store and process massive datasets.
- **Performance:** Databases optimize data access and retrieval. MongoDB uses indexing to improve query performance, allowing you to find specific documents quickly.
- **Durability:** Databases ensure data persistence. MongoDB offers features like replication and backups to protect your data against hardware failures or accidental deletion.

- **Concurrency Control:** Databases manage concurrent access to data. MongoDB provides mechanisms to prevent conflicts when multiple applications or users try to modify the same data simultaneously.

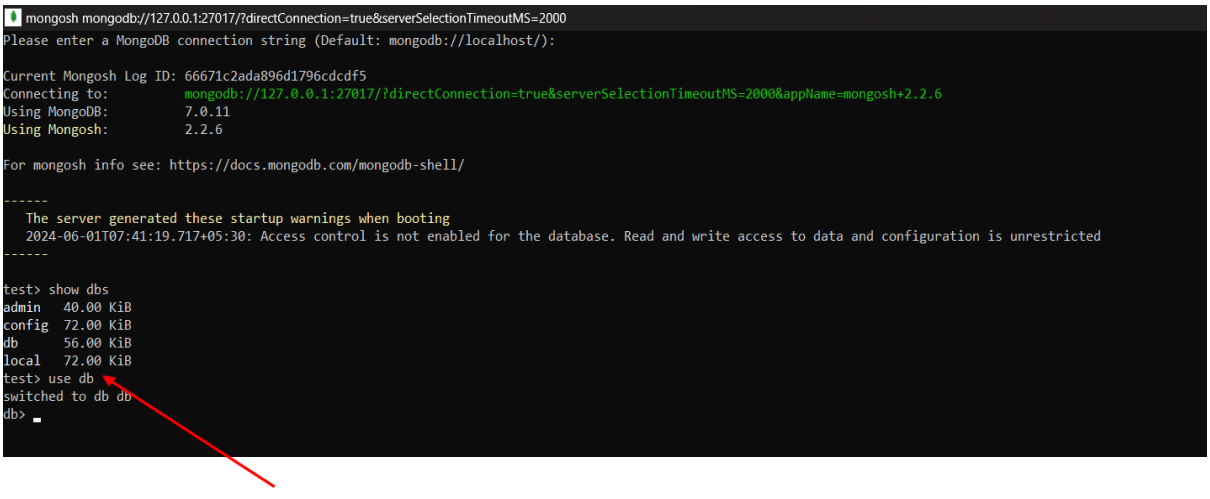
Using the MongoDB Shell:

The MongoDB shell is a command-line interface that allows you to interact with your MongoDB server directly. You can use the shell to create and manage databases, collections, and documents. Here are some basic shell commands:

- **use <database_name>:** Switch to a specific database.
- **show dbs:** List all available databases.



Show dbs: On writing command show dbs in MongoShell **"admin", "config", "db" ,"local"** are the **database.**



Use db:switches you to a specific database in the MongoDB shell.
This tells MongoDB which database to use for subsequent commands like creating collections or finding documents.
Multiple databases can exist on a MongoDB server, allowing you to organize data logically.

COLLECTIONS:

Collections are just like tables in relational databases, they also store data, but in the form of documents. A single database is allowed to store multiple collections.


```
_id: ObjectId('66570191c2173a0885516126')
name : "Student 948"
age : 19
courses : "['English', 'Computer Science', 'Physics', 'Mathematics']"
gpa : 3.44
home_city : "City 2"
blood_group : "O+"
is_hotel_resident : true
```

```
_id: ObjectId('66570191c2173a0885516127')
name : "Student 157"
age : 20
courses : "['Physics', 'English']"
gpa : 2.27
home_city : "City 4"
blood_group : "O-"
is_hotel_resident : true
```

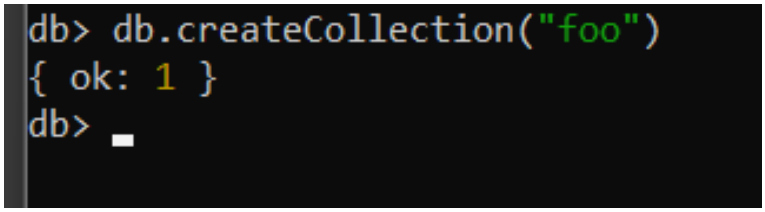

As we know that MongoDB databases are schemaless. So, it is not necessary in a collection that the schema of one document is similar to another document. Or in other words, a single collection contains different types of documents like as shown in the below example where **"Students"** collection contain two different types of documents

Using Collections:

There are two main ways to interact with collections in MongoDB:

1. **Using the MongoDB Shell:**
   - **db.createCollection("<collection_name>"):** Creates a new collection within the current database.
   - **db.<collection_name>.insertOne({<key1>: <value1>, ...}):** Inserts a new document into a collection.
   - **db.<collection_name>.find({}):** Finds all documents in a collection.

Example for above commands:



Here, We have created an collection named **"foo".**

```
db> db.foo.insertOne({"name":"Shreya","Dept":"Data Science"})
{
  acknowledged: true,
  insertedId: ObjectId('666727cfda896d1796cdcdf6')
}
db> db.foo.find({})
[
  {
    _id: ObjectId('666727cfda896d1796cdcdf6'),
    name: 'Shreya',
    Dept: 'Data Science'
  }
]
```

To the created collection we have inserted an **key and value** in the above example and we have **find** the data what we have inserted.

Naming Restrictions for Collection:

- Before creating a collection you should first learn about the naming restrictions for collections:
- Collection name must starts with an underscore or a character.
- Collection name does not contain $, empty string, null character and does not begin with system. prefix.
- The maximum length of the collection name is 120 bytes(including the database name, dot separator, and the collection name).

DOCUMENTS:

In MongoDB, the data records are stored as BSON documents. Here, BSON stands for binary representation of JSON documents, although BSON contains more data types as compared to JSON. The document is created using field-value pairs or key-value pairs and the value of the field can be of any BSON type.

{

field1: value1

field2: value2

....

fieldN: valueN

}

Documents are the foundation of data storage in MongoDB. Unlike relational databases with rigid table structures, MongoDB offers a flexible approach using schema-less documents. These documents resemble JSON objects, making them easy to understand and work with.

Structure of a Document:

A MongoDB document is built using key-value pairs enclosed in curly braces {}. Each key acts as a field name (similar to column names in relational databases) and holds a corresponding value. These values can be of various data types, including:

Strings

Numbers

Booleans

Dates

Arrays (ordered collections of values)

Embedded Documents (documents within documents)

Example 2: Product Information

Consider an e-commerce application. You can store product information in documents within a collection named products:

```
{
  "_id": ObjectId("507f191e810c19729de8600d"),
  "name": "T-Shirt",
  "price": 19.99,
  "description": "A comfortable and stylish T-shirt",
  "category": "Clothing",
  "size": ["S", "M", "L"],
  "colors": ["Red", "Blue", "Black"],
  "inventory": {
    "S": 10,
    "M": 20,
    "L": 5
  }
}
```

Here, the product document demonstrates:

Arrays: The size and colors fields use arrays to store multiple values.

Embedded Document: The inventory field is an embedded document that holds stock information for each size.

DATATYPES:

MongoDB offers a rich set of data types to represent various kinds of information within your documents. These data types provide structure and ensure data integrity within your MongoDB collections. Here's a detailed explanation of some common data types in MongoDB (along with their BSON equivalents, as BSON is the internal binary format used by MongoDB to store data):

Basic Data Types:

- **String (BSON: string):** Represents textual data. This is the most commonly used data type for storing names, descriptions, addresses, etc.
- **Number (BSON: double):** Stores numerical values, including integers, floating-point numbers, and decimals. You can use this for quantities, prices, ratings, etc.
- **Boolean (BSON: bool):** Represents Boolean values (true or false). This is useful for flags, on/off switches, or any binary decision points in your data.
- **Date (BSON: date):** Stores date and time information. This can be used for timestamps, creation dates, or any time-related data.
- **ObjectId (BSON: objectId):** A special data type in MongoDB that automatically generates a unique 12-byte identifier for each document. This is often used as the primary key for documents within a collection.

Complex Data Types:

- **Array (BSON: array):** Represents an ordered collection of values. You can use arrays to store lists of items, tags, categories, or any scenario where you have multiple related values for a single field.
- **Null (BSON: null):** Indicates the absence of a value for a specific field. This can be helpful when data might be missing or not applicable for a particular document.

Advanced Data Types:

- **Timestamp (BSON: timestamp):** Stores a timestamp with millisecond precision. This is useful for capturing exact moments in time beyond just dates.
- **Binary Data (BSON: binary):** Used to store raw binary data like images, audio files, or any other non-textual data. There's a size limitation for binary data stored within a document.
- **Regular Expression (BSON: regex):** Represents a pattern for matching text strings. This can be useful for implementing search functionality based on patterns within your data.

- **Code (BSON: code):** Allows you to store JavaScript code snippets within documents. This is rarely used but can be helpful for implementing custom logic within the database.
- **DBPointer (BSON: dbPointer):** A reference to another document within the same database. This allows you to create relationships between documents across collections.
- **Decimal128 (BSON: decimal128):** Stores high-precision decimal numbers. This is useful for financial calculations or scientific applications requiring precise monetary values.

Choosing the Right Data Type:

Selecting the appropriate data type for each field in your documents is crucial. Here are some tips:

- Use strings for textual data.
- Use numbers for numerical values (consider integers or decimals depending on precision needs).
- Use booleans for true/false flags.
- Use dates for storing timestamps.
- Use arrays for lists of related values.
- Use ObjectIds for unique document identifiers.