To run MongoDB in Docker, you can use the command-line interface (CLI) for a single container or Docker Compose for more complex applications. Using Docker is especially beneficial for development and testing, as it creates a consistent and portable database environment.

**Prerequisites**

* You must have Docker installed on your system. You can verify this by running docker --version in your terminal.

**Method 1: Use docker run for a basic container**

This method is suitable for quickly starting a single MongoDB instance for development or testing.

**1. Create a Docker volume for data persistence**

By default, data inside a container is deleted when the container is removed. To save your data permanently, create a named volume.

docker volume create mongo\_data

**2. Run the MongoDB container**

Use the docker run command to create and start the container in the background.

docker run --name mongodb-container -d -p 27017:27017 -v mongo\_data:/data/db mongo

* --name mongodb-container: Assigns a name to your container for easy reference.
* -d: Runs the container in detached (background) mode.
* -p 27017:27017: Maps port 27017 from the host machine to port 27017 in the container, allowing external connections.
* -v mongo\_data:/data/db: Mounts the named volume mongo\_data to the default MongoDB data directory /data/db inside the container.
* mongo: Specifies that you want to use the official MongoDB image from Docker Hub.

**3. Verify the container is running**

Check the status of your Docker containers to confirm MongoDB is active.

docker ps

**4. Access the MongoDB shell**

To interact with your running database, execute the mongosh command inside the container.

docker exec -it mongodb-container mongosh

**Method 2: Use Docker Compose for multiple services**

Docker Compose is ideal for defining and running multi-container applications, such as a backend server and a database. This is the best approach for managing an application with persistent storage and a secure setup.

**1. Create a docker-compose.yml file**

Create a file named docker-compose.yml in your project directory and add the following content. This configuration sets up a secure MongoDB container with a persistent volume.

Docker-compose.yaml

services:

mongodb:

image: mongo:latest

container\_name: mongodb-container

ports:

- "27017:27017"

environment:

- MONGO\_INITDB\_ROOT\_USERNAME=admin

- MONGO\_INITDB\_ROOT\_PASSWORD=password

volumes:

- mongodb-data:/data/db

restart: unless-stopped

volumes:

mongodb-data:

* image: mongo:latest: Specifies the image to use.
* container\_name: mongodb-container: Sets a friendly name for the container.
* ports: Exposes MongoDB to your host machine.
* environment: Sets the root username and password for your database, a crucial step for a secure setup.
* volumes: Defines a named volume for persistent storage.
* restart: unless-stopped: Ensures the container restarts automatically if it crashes.

**2. Start your services**

From the same directory as your docker-compose.yml file, run the following command to start MongoDB in the background.

docker compose up -d

**3. Access the MongoDB shell**

Connect to the database using the MongoDB shell with the username and password you configured.

docker exec -it mongodb-container mongosh -u admin -p password --authenticationDatabase admin

(Note: Make sure the container is running before executing the “docker exec” command)

Once inside the mongosh shell, you can run the following commands.

**Database operations**

* show dbs: Lists all databases on the [MongoDB server](https://www.mongodb.com/).
* use <database\_name>: Switches to the specified database, creating it if it doesn't exist.
* db.stats(): Provides statistics for the current database.

**Collection operations**

* db.createCollection('<collection\_name>'): Creates a new collection in the current database.
* show collections: Lists all collections in the current database.
* db.<collection\_name>.drop(): Deletes the specified collection.

**Document (CRUD) operations**

* db.<collection\_name>.insertOne({ ... }): Inserts a new document into the collection.
* db.<collection\_name>.find(): Retrieves all documents in a collection.
* db.<collection\_name>.find({ key: 'value' }): Searches for documents matching the specified filter.
* db.<collection\_name>.updateOne({ ... }, { ... }): Updates a single document in the collection.
* db.<collection\_name>.deleteOne({ ... }): Deletes a single document from the collection.

**Exit the shell**

* exit: Closes the MongoDB shell session and returns you to your local command line.

Performing CRUD operation using Mongosh (inside docker)

**Prerequisites**

* You must be inside the mongosh shell. If you are running MongoDB in Docker with authentication, connect with the following command:

docker exec -it mongodb-container mongosh -u admin -p password --authenticationDatabase admin

**Step 1: Select a database**

To perform operations, you must first specify which database you want to work with.

use store

*Note: If the store database does not exist, MongoDB will create it automatically upon the first data insertion.*

**Create (Insert) Operations**

The create operations add new documents to a collection.

**insertOne(): Insert a single document**

db.products.insertOne(

{

name: "Laptop",

price: 1200,

category: "Electronics",

inStock: true

}

)

**insertMany(): Insert multiple documents**

You can insert an array of documents at once.

db.products.insertMany([

{

name: "T-Shirt",

price: 25,

category: "Apparel",

inStock: true

},

{

name: "Coffee Mug",

price: 15,

category: "Home",

inStock: false

}

])

**Read (Query) Operations**

Read operations retrieve documents from a collection based on specified criteria.

**find(): Retrieve all documents**

To view all documents in a collection, pass an empty query filter {}.

db.products.find({})

For a more readable, formatted output, chain the .pretty() method.

db.products.find({}).pretty()

**find() with a filter: Find documents that match a condition**

To find documents that match a specific condition, provide a query filter.

*// Find products with a price greater than 1000*

db.products.find({ price: { $gt: 1000 } })

*// Find products in the 'Apparel' category*

db.products.find({ category: "Apparel" })

**findOne(): Retrieve a single document**

This command returns the first document that matches the specified filter.

db.products.findOne({ name: "Laptop" })

**Update Operations**

Update operations modify existing documents in a collection.

**updateOne(): Update a single document**

This method updates the first document that matches the filter.

db.products.updateOne(

{ name: "Laptop" }, *// Filter for the document*

{ $set: { price: 1150, inStock: false } } *// Update operator*

)

* The $set operator updates the value of a field. If the field does not exist, it creates it.

**updateMany(): Update multiple documents**

This method updates all documents that match the specified filter.

db.products.updateMany(

{ category: "Apparel" },

{ $set: { inStock: true } }

)

**replaceOne(): Replace a single document**

This method replaces the entire document, except for the \_id field.

db.products.replaceOne(

{ name: "Coffee Mug" },

{

name: "Travel Mug",

price: 30,

category: "Home & Kitchen",

inStock: true

}

)

**Delete Operations**

Delete operations remove existing documents from a collection.

**deleteOne(): Delete a single document**

This deletes at most one document matching the specified filter.

db.products.deleteOne({ name: "Travel Mug" })

**deleteMany(): Delete multiple documents**

This deletes all documents that match the specified filter.

*// Delete all products in the 'Home' category*

db.products.deleteMany({ category: "Home" })

*// Delete all documents in the collection*

db.products.deleteMany({})

**drop(): Delete the entire collection**

For removing all documents from a large collection, dropping and recreating it is more efficient than using deleteMany() with an empty filter.

db.products.drop()