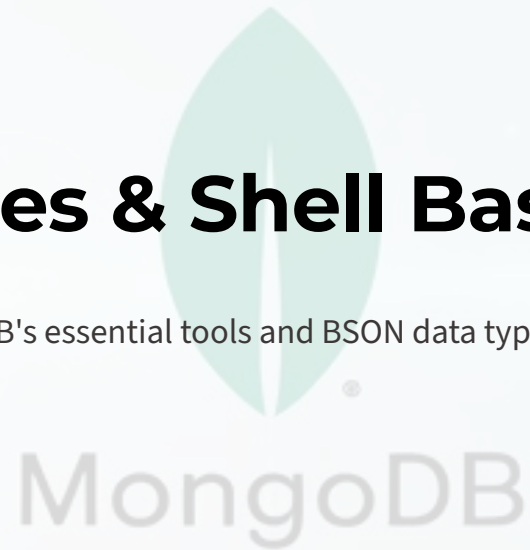


# MongoDB Data Types & Shell Basics

A comprehensive guide for beginners learning MongoDB's essential tools and BSON data types. Updated October 2025.



# What Is MongoDB?

MongoDB is a **NoSQL, document-oriented database** that stores data in flexible, JSON-like documents rather than traditional rows and columns. This flexibility allows you to adapt your data structure as your application evolves, without the rigid schema requirements of relational databases.

Unlike SQL databases that require predefined tables, MongoDB collections can contain documents with varying structures. This makes it ideal for modern applications that handle diverse or rapidly changing data.

# Two Essential Tools for Working with MongoDB

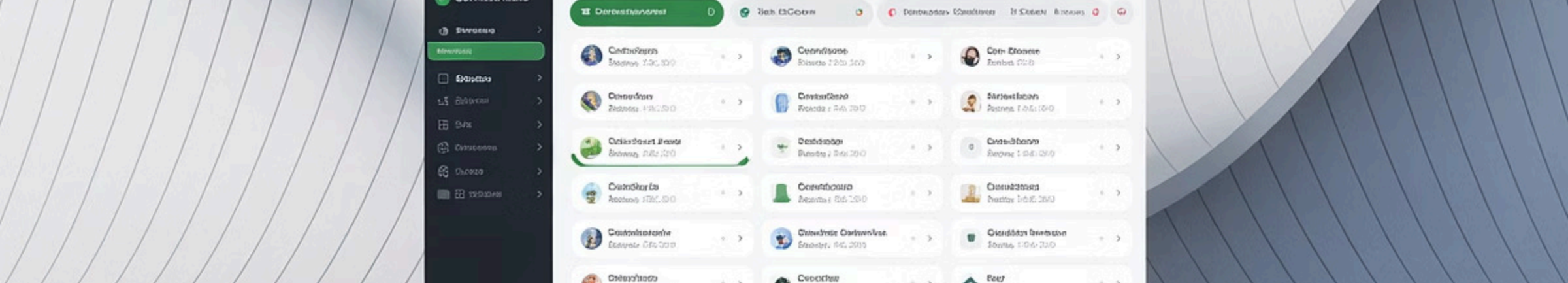
## **Mongosh (MongoDB Shell)**

A powerful command-line interface for scripting, running queries, and performing database operations. Perfect for automation, testing, and advanced database management tasks.

## **MongoDB Compass**

An intuitive graphical user interface that lets you visually explore databases, collections, and documents. Ideal for beginners who prefer point-and-click interactions over command-line work.

Both tools work with the same MongoDB databases, so you can choose whichever suits your workflow. Many developers use both: Compass for exploration and Mongosh for scripting.



# Getting Started with MongoDB Compass

When you launch Compass, you'll see a clean interface displaying all your databases in the left sidebar. Click on any database to reveal its collections, then select a collection to browse its documents.

The visual interface shows document structures clearly, with expandable fields and colour-coded data types. This makes it easy to understand your data's shape at a glance, especially when you're just starting out with MongoDB.

# Using the MongoDB Shell (Mongosh)

## Interactive Command Line

Launch mongosh from your terminal to access an interactive prompt where you can execute JavaScript commands directly. The shell provides immediate feedback, making it excellent for testing queries and exploring data structures.

Type commands, press Enter, and see results instantly. The shell supports full JavaScript syntax, allowing you to use variables, functions, and loops.



# Understanding BSON Data Types

MongoDB uses **BSON** (Binary JSON) to store documents. BSON extends JSON with additional data types needed for database operations. Understanding these types is crucial for effective data modelling.

| Type           | Syntax       | Example                              |
|----------------|--------------|--------------------------------------|
| String         | "text"       | "hello world"                        |
| Object         | {}           | {name: "John", age: 30}              |
| Array          | []           | [1, 2, 3, "four"]                    |
| Boolean        | true/false   | true                                 |
| 32-bit Integer | NumberInt()  | NumberInt(42)                        |
| 64-bit Integer | NumberLong() | NumberLong("9223372036854775807")    |
| Double         | 0.0          | 5.75                                 |
| Date           | ISODate()    | ISODate("2025-10-20")                |
| ObjectId       | ObjectId()   | ObjectId("507f1f77bcf86cd799439011") |

[illegible]

| Plain Numbers  | Explicit Integers   |
|--|---|
| When you enter 10, it displays as 10 (not 10.0). Decimal numbers like 5.75 stay as 5.75. | NumberInt(10) displays as Int32(10), clearly showing it's a 32-bit integer. |

## Long Integers

`NumberLong("123")` displays as `Long('123')`. Always use string arguments!

## Long Integers

`NumberLong("123")` displays as `Long('123')`. Always use string arguments!

## Long Integers

`NumberLong("123")` displays as `Long('123')`. Always use string arguments!

## Long Integers

`NumberLong("123")` displays as `Long('123')`. Always use string arguments!

## Long Integers

`NumberLong("123")` displays as `Long('123')`. Always use string arguments!

## Long Integers

`NumberLong("123")` displays as `Long('123')`. Always use string arguments!

# Important: NumberLong Syntax Update

## **Deprecated Syntax**

The old syntax `NumberLong(123)` using a number argument is now deprecated. Modern MongoDB requires string arguments.

## **Don't Use**

```
NumberLong(123)  
// Deprecated - may cause errors
```

## **Always Use**

```
NumberLong("123")  
// Correct modern syntax
```

This change ensures consistency and prevents potential data loss when working with very large numbers that exceed JavaScript's number precision limits.

# Checking Variable Types in the Shell

MongoDB shell supports JavaScript operators for type inspection. These tools help you verify your data structures during development and debugging.

```
var obj = {
  a: "hello",
  b: {nested: true},
  c: [1, 2, 3],
  d: true
};

// Display the object
print(obj);

// Check individual types
typeof obj.a // "string"
typeof obj.b // "object"
obj.c instanceof Array // true
typeof obj.d // "boolean"
```

# Compass

{ } My Queries

## CONNECTIONS (1)

### Local MongoDB

admin

config

local

test

test

test

mongosh: Local Mongo...



>\_MONGOSH



```
> use test
< already on db test
> var obj = {a : "" , b : {} , c : [] , d : true};
> print(obj);
< { a: '', b: {}, c: [], d: true }
> typeof obj
< object
> typeof obj.a
< string
> typeof obj.b
< object
> typeof obj.c
< object
> typeof obj.d
< boolean
> obj.c instanceof Array;
< true
test>
```

# ObjectId: MongoDB's Unique Identifier

Every MongoDB document requires a unique `_id` field. If you don't provide one, MongoDB automatically generates an **ObjectId**. Understanding ObjectIds helps you work with document relationships and troubleshoot data issues.

01

---

## Timestamp Component

First 4 bytes store creation time (seconds since Unix epoch)

02

---

## Random Value

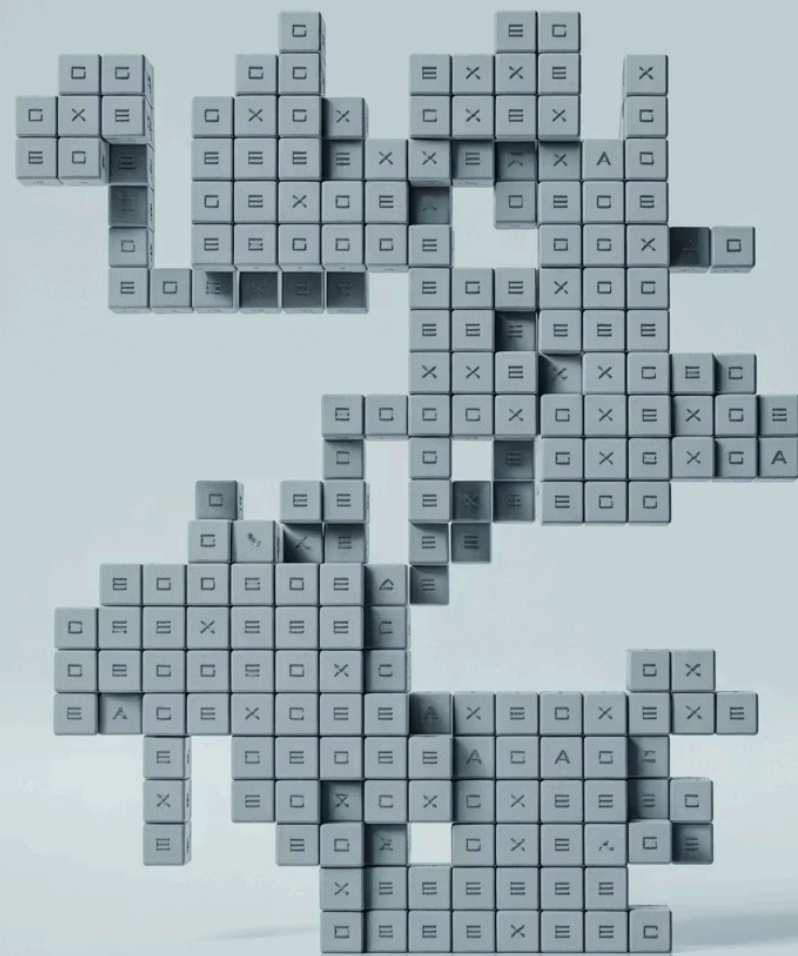
Next 5 bytes combine machine identifier and process ID

03

---

## Counter

Final 3 bytes contain a random incrementing counter



```
> ObjectId()  
< ObjectId('68f612557639ae8acefb9590')  
> ObjectId()  
< ObjectId('68f612797639ae8acefb9591')  
> ObjectId()  
< ObjectId('68f6127a7639ae8acefb9592')  
> ObjectId()  
< ObjectId('68f6127b7639ae8acefb9593')  
> ObjectId()  
< ObjectId('68f6127d7639ae8acefb9594')  
test>
```

# Creating and Using ObjectIds

## Generate New ObjectId

```
ObjectId()  
// Creates: ObjectId("67a2b8f...")  
  
// Use in documents  
db.users.insertOne({  
  _id: ObjectId(),  
  name: "Alice"  
})
```

## Reference Specific ObjectId

```
ObjectId('507f1f77bcf86cd799439011')  
// Uses exact ID provided  
  
// Query by ObjectId  
db.users.findOne({  
  _id: ObjectId('507f...')  
})
```

ObjectIds are globally unique across collections and databases. The embedded timestamp makes them sortable by creation time, which is useful for chronological queries.



# Working with Dates in MongoDB

MongoDB provides robust date handling through BSON date types. Proper date storage ensures accurate queries and sorting by time.

1

## JavaScript Date

```
new Date("2025-10-20")
```

Creates JS Date object, displays as ISO string

2

## BSON ISODate

```
ISODate("2025-10-20")
```

Native BSON date type for database storage

3

## Stored Format

Both stored as milliseconds since epoch for precise queries

# Compass

{ } My Queries

## CONNECTIONS (1)

### Local MongoDB

admin

config

local

test

test

test

mongosh: Local Mongo...

mongosh: Local Mongo...



>\_MONGOSH



```
> use test
< already on db test
> Date
< [Function: Date]
> Date()
< Mon Oct 20 2025 16:17:58 GMT+0530 (India Standard Time)
> new Date()
< 2025-10-20T10:48:20.443Z
> ISODate()
< 2025-10-20T10:49:01.243Z
> ISODate
< [Function: ISODate] { help: [Function (anonymous)] Help }
test> |
```

# Compass

{ } My Queries

## CONNECTIONS (1)

Search connections

▼ Local MongoDB

▶ admin

▶ config

▶ local

▼ test

test

test

> mongosh: Local Mongo...

> mongosh: Local Mongo...

>\_MONGOSH

> use test

< already on db test

> new Date("2025 October 20")

< 2025-10-19T18:30:00.000Z

> new Date("2025-10-20")

< 2025-10-20T00:00:00.000Z

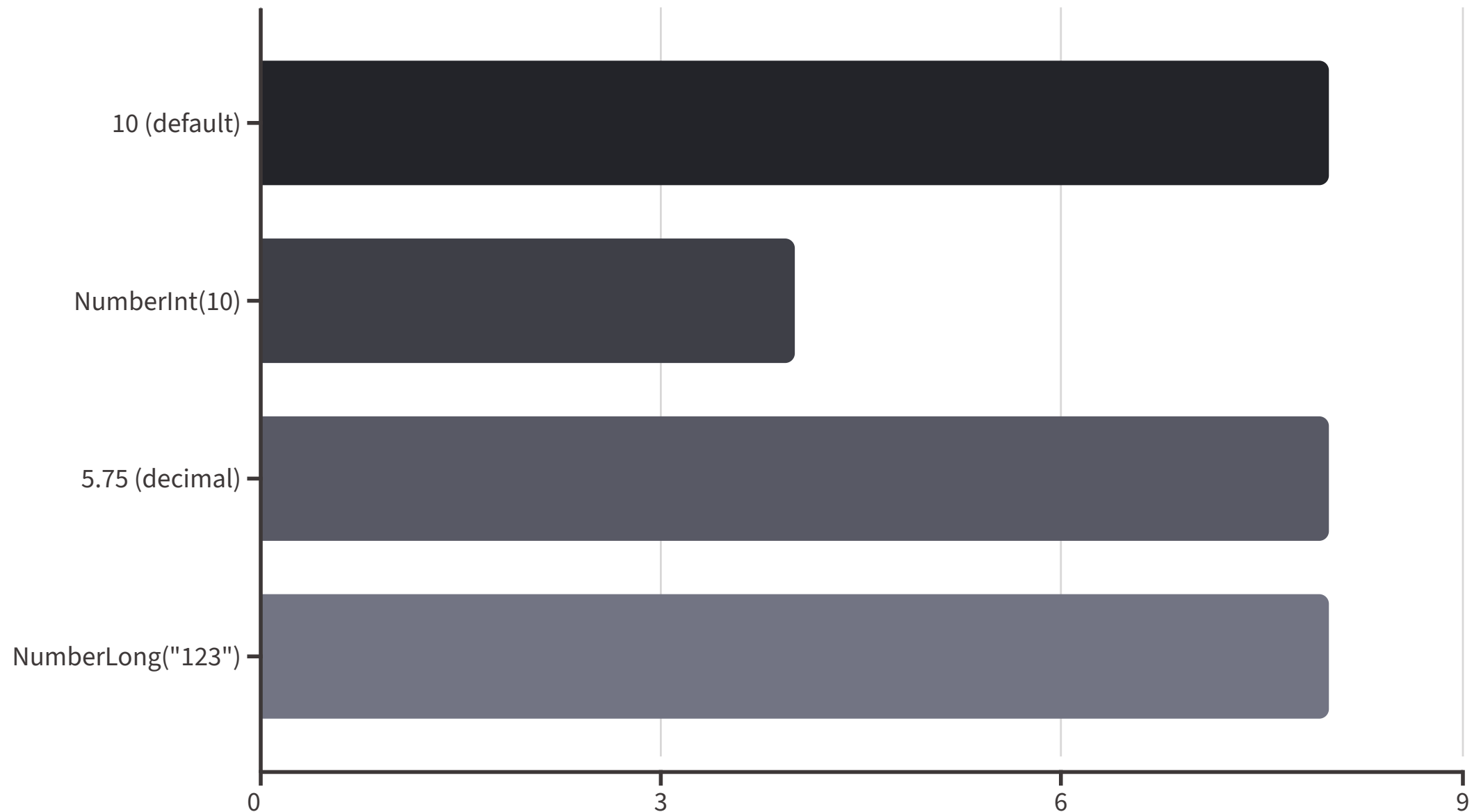
> new Date("2025,10,20")

< 2025-10-19T18:30:00.000Z

test>

# Number Storage: The Double Default

By default, MongoDB stores all numbers as **64-bit floating-point doubles** unless you explicitly specify an integer type. This behaviour differs from traditional programming languages and affects storage efficiency.



Use explicit integer types when storing whole numbers to save storage space and improve query performance, especially for large collections.

# Compass

{ } My Queries

## CONNECTIONS (1)

### Local MongoDB

admin

config

local

test

test

>\_MONGOSH

> use test

< already on db test

> obj = {a : 10 , b : 0 , c : 5.75 , d : NumberInt(10) , e : NumberLong(123)}

< Warning: NumberLong: specifying a number as argument is deprecated and may lead to loss of precisi

< { a: 10, b: 0, c: 5.75, d: Int32(10), e: Long('123') }

> obj

< { a: 10, b: 0, c: 5.75, d: Int32(10), e: Long('123') }

> typeof obj.a

< number

> typeof obj.b

< number

> typeof obj.c

< number

> typeof obj.d

< object

> typeof obj.f

< undefined

> typeof obj.e

< object

> NumberLong("123")

< Long('123')

> NumberInt(10)

< Int32(10)

> obj = {a : 10 , b : 0 , c : 5.75 , d : NumberInt(10) , e : NumberLong('123')}

< { a: 10, b: 0, c: 5.75, d: Int32(10), e: Long('123') }

test>

# Best Practices for Data Types

1

## Choose Appropriate Types

Use `NumberInt()` for counters and small integers, `NumberLong()` for IDs and large numbers, and `Double` for precise decimals. Type choice affects storage and performance.

2

## Always Use String Arguments

When calling `NumberLong()` or `NumberInt()`, wrap values in quotes: `NumberLong("123")`. This prevents precision loss and follows current MongoDB standards.

3

## Leverage ObjectIds

Let MongoDB generate `ObjectIds` for document identity. The embedded timestamp and uniqueness guarantees make them ideal primary keys without additional indexing.

4

## Store Dates Consistently

Always use `ISODate()` for date values. This ensures proper timezone handling, accurate sorting, and enables powerful date range queries across your application.



# Practice Challenge

## Put Your Knowledge to the Test

Open MongoDB Compass or mongosh and create a new document that includes each major BSON type we've covered. Then use `typeof` and `instanceof` to verify each field's type.

### Create a complex object

Include strings, numbers (both Double and NumberInt), arrays, nested objects, booleans, a date, and an ObjectId

### Inspect the types

Use shell commands to check each field's type and verify it matches your expectations

### Experiment with queries

Try finding documents by different field types and see how MongoDB handles type-specific operations

This hands-on practice will solidify your understanding of MongoDB's type system and prepare you for real-world data modelling challenges.