Lesson:

Data Types in MongoDB

PW SKILLS

Topics

- Introduction to Data Types in MongoDB
- Common Data Types in MongoDB
- Advanced-Data Types in MongoDB
- Schema Design Considerations for Data Types
- Handling Null and Undefined Values
- Data Type Conversion in MongoDB
- Regular Expressions with String Data
- Interview points

Introduction to Data Types in MongoDB

MongoDB, a NoSQL database, stores data in a flexible, JSON-like format called BSON (Binary JSON). Understanding the various data types MongoDB supports is crucial for effective data modeling and querying. In this article, we'll explore the fundamental concepts behind MongoDB data types.

MongoDB offers a diverse set of data types, including but not limited to integers, strings, arrays, and documents. These data types enable developers to represent complex structures efficiently. The flexibility of MongoDB's schema allows for dynamic and nested data, making it suitable for a variety of applications.

Common Data Types in MongoDB

MongoDB offers a variety of data types to accommodate the dynamic and diverse nature of modern applications. In this article, we'll delve into some of the common data types in MongoDB, including String, Integer, Double, Boolean, and Date.

We will walk through the common data types by taking the example of an E-com database with a collection of product

Firstly, create the database using MongoDB shell using the command below -

```
Unset
test>use e-com
e-com>
```

ObjectId

In MongoDB, an ObjectId is a 12-byte identifier typically employed as the default identifier for documents within a collection. It is a BSON (Binary JSON) data type specifically designed to be lightweight, fast to generate, and globally unique.
For example - When we insert data (name: “cabbage”) to the e-com database, the objectId will be automatically generated.

i.e.

```
Unset
e-com> db.products.insertOne({name: "cabbage"})
{
  acknowledged: true,
  insertedId: ObjectId("65a0e796060faf791d53dc4a")
}
e-com> db.products.find({name: "cabbage"})
[ { _id: ObjectId("65a0e796060faf791d53dc4a"), name: 'cabbage' } ]
e-com>
```

String

The string data type in MongoDB is used for storing textual information. It can represent anything from names and descriptions to complex text documents. MongoDB provides various operators and features to perform efficient string manipulations, making it a powerful tool for handling diverse text-based requirements.

For example, Let’s take the “name” and “category” fields to demonstrate the use of the String data type to store textual information
i.e

```
Unset
e-com> db.products.insertOne({name:"tomato", category: "Vegetables"})
{
  acknowledged: true,
  insertedId: ObjectId("659fc99c8022ec1ecd7a5524")
}

e-com> db.products.find().pretty()
[
  {
    _id: ObjectId("659fc99c8022ec1ecd7a5524"),
    name: 'tomato',
    category: 'Vegetables'
  }
]
e-com>
```

Integer

The Integer data type is straightforward, representing whole numbers without any decimal points. It's ideal for fields where only whole numeric values are applicable, such as quantities, counts, or identifiers.

For example, Let's add the "quantity" fields to showcase the Integer data type, representing whole numbers without decimal points.
i.e

```
Unset
e-com> db.products.updateOne({ name: "tomato" },{ $set: { quantity: 20 } } )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
e-com> db.products.find()
[


  {
    _id: ObjectId("659fc99c8022ec1ecd7a5524"),
    name: 'tomato',
    category: 'Vegetables',
    quantity: 20
  }
]
```

Note - the "updateOne" is a command used for updating collections in the MongoDB database

Double

For scenarios that require numeric values with decimal points, MongoDB employs the Double data type. This is particularly useful when dealing with measurements, financial data, or any situation where precision matters.

For example, let's add the "price" and "discountPercentage" fields to illustrate the Double data type for handling numeric values with decimal points.
i.e

```
Unset
e-com> db.products.updateOne( { name: "tomato" }, { $set: { price: 99.99 }} )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}

e-com> db.products.find()
[
  {
    _id: ObjectId("659fc99c8022ec1ecd7a5524"),
    name: 'tomato',

     category: 'Vegetables',
     quantity: 20,
     price: 99.99
   }
 ]
```

Boolean

Boolean data type is fundamental for representing true or false values. It's commonly used for binary decisions, status indicators, or any situation where data can be categorized as either on or off.

For example, let's add the"stockEmpty" field using the Boolean data type, representing true or false values.
i.e

```
Unset
e-com> db.products.updateOne( { name: "tomato" }, { $set: { stockEmpty: false
}} )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}

e-com> db.products.find()
[
  {
    _id: ObjectId("659fc99c8022ec1ecd7a5524"),
    name: 'tomato',
    category: 'Vegetables',
    quantity: 20,
    price: 99.99,
    stockEmpty: false
  }
]
e-com>
```

Date

The date data type in MongoDB is designed for handling dates and times. Storing timestamps accurately is crucial in many applications, and MongoDB's Date type simplifies the process. It allows for easy sorting, querying, and manipulation of temporal data.

For example, Let's add "lastModified" field to demonstrate the Date data.

```
Unset
e-com> db.products.updateOne( { name: "tomato" }, { $set: { lastModified: new
Date()}} )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
e-com> db.products.find()
[
  {
    _id: ObjectId("659fc99c8022ec1ecd7a5524"),
    name: 'tomato',
    category: 'Vegetables',
    quantity: 20,
    price: 99.99,
    stockEmpty: false,
    lastModified: ISODate("2024-01-11T11:36:17.956Z")
  }
]
e-com>
```

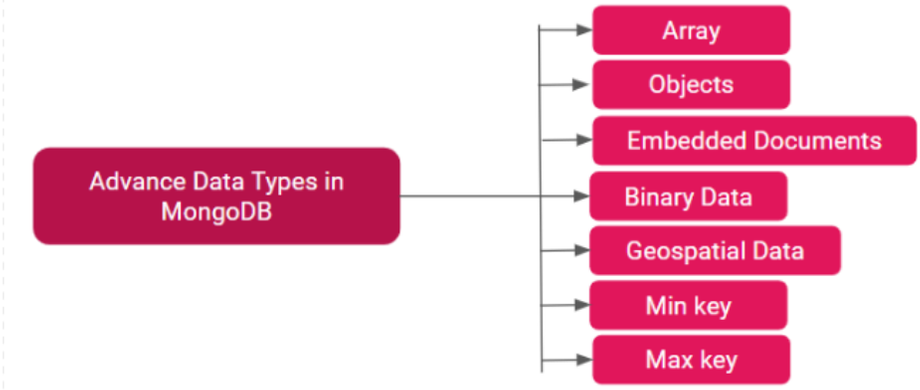
Timestamp

In MongoDB, a timestamp is often associated with the ObjectId type. The timestamp is a 4-byte value that represents the number of seconds since the Unix epoch (January 1, 1970, at 00:00:00 UTC). It is an integral part of the ObjectId structure and serves several purposes, mainly related to the generation of unique identifiers.

For example, we can insert data along with a Timestamp constructor i.e

```
Unset
e-com> db.products.insertOne({name: "potato", ts : new Timestamp()})
{
  acknowledged: true,
  insertedId: ObjectId("65a0f4ba060faf791d53dc4b")
}
e-com> db.products.findOne({name:"potato"})
{
  _id: ObjectId("65a0f4ba060faf791d53dc4b"),
  name: 'potato',
  ts: Timestamp({ t: 1705047226, i: 1 })
}
e-com>
```

Advanced-Data Types in MongoDB



MongoDB, a leading NoSQL database, goes beyond basic data types, offering advanced features to handle complex data structures. In this article, we'll delve into four advanced data types: Array, Objects, Embedded Documents, and Binary Data. Understanding these data types opens up possibilities for designing sophisticated and flexible MongoDB schemas.

We will be using the same database i.e e-com created above to illustrate the following data types with example

Array

Arrays in MongoDB provide a powerful way to store lists of values within a single document field. Whether it's a collection of tags, comments, or related items, arrays allow for efficient storage and retrieval of multiple values. We'll explore the various array operators and considerations for optimizing array-based queries.

For example, we can add an array of tags that contain fresh, green, natural, and salad.
i.e

```
Unset
e-com> db.products.updateOne( { name: "tomato" }, { $set: { tags: ["fresh",
"green", "natural", "salad"]}} )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}

e-com> db.products.find()
[
  {
    _id: ObjectId("659fc99c8022ec1ecd7a5524"),
    name: 'tomato',
    category: 'Vegetables',
    quantity: 20,
    price: 99.99,
    stockEmpty: false,
    lastModified: ISODate("2024-01-11T11:36:17.956Z"),
    tags: [ 'fresh', 'green', 'natural', 'salad' ]
  }
]
e-com>
```

Objects

Objects, also known as subdocuments, enable the creation of hierarchical structures within MongoDB documents. This allows for the representation of complex relationships and nested data.

For example, we can add an object of meta-data consisting of the product provider name, source, and contact number
i.e

```
Unset
e-com> db.products.updateOne( { name: "tomato" }, { $set: {productOwner: {name:
"john doe", source: "bangalore", contact: "8726547861"} }} )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
e-com> db.products.find()
[
  {
    _id: ObjectId("659fc99c8022ec1ecd7a5524"),
    name: 'tomato',
    category: 'Vegetables',
    quantity: 20,
    price: 99.99,
    stockEmpty: false,
    lastModified: ISODate("2024-01-11T11:36:17.956Z"),
    tags: [ 'fresh', 'green', 'natural', 'salad' ],
    productOwner: { name: 'john doe', source: 'bangalore', contact:
'8726547861' }
  }
]
e-com>
```

Embedded Documents and

Embedded documents take the concept of objects further by allowing entire documents to be nested within other documents. This provides a natural way to model relationships between entities and reduces the need for separate collections.

For example, let's add a “reviews” field that contains nested documents of “index” (1, 2)

```
Unset
e-com> db.products.updateMany( { name: "tomato" }, { $set: {reviews: { title: "users
Reviews", 1: {_id: "1234", review: "good taste"}, 2: {_id: "1235", review: "cheap and
tasty"}}  }})
{
 acknowledged: true,
 insertedId: null,
 matchedCount: 1,
 modifiedCount: 1,
 upsertedCount: 0
}
e-com> db.products.find()
[
  {
    _id: ObjectId("659fc99c8022ec1ecd7a5524"),
    name: 'tomato',
    category: 'Vegetables',
    quantity: 20,
    price: 99.99,
    stockEmpty: false,
    lastModified: ISODate("2024-01-11T11:36:17.956Z"),
    tags: [ 'fresh', 'green', 'natural', 'salad' ],
    productOwner: { name: 'john doe', source: 'bangalore', contact:
'8726547861' },
    reviews: {
      '1': { _id: '1234', review: 'good taste' },
      '2': { _id: '1235', review: 'cheap and tasty' },
      title: 'users Reviews'
    }
  }
]
e-com>
```

Binary Data

Binary data type in MongoDB allows the storage of binary information, such as images, documents, or serialized objects. Leveraging the binary data type enables efficient handling of non-textual information, offering a versatile solution for diverse data storage needs.

For example, Let's add the "image" field that contains binary data encoded in Base64, by using the BinData constructor for demonstration purposes. MongoDB provides powerful geospatial queries, allowing you to find documents within a specific distance or shape.

```
Unset
e-com> db.products.updateOne( { name: "tomato" }, { $set: {image: {name: new
BinData(0, "aGVsbG8gd29ybGQ=") }}} )
{
 acknowledged: true,
 insertedId: null,
 matchedCount: 1,
 modifiedCount: 1,
 upsertedCount: 0
}
e-com> db.products.find()
[
 {
  _id: ObjectId("659fc99c8022ec1ecd7a5524"),
  name: 'tomato',
  category: 'Vegetables',
  quantity: 20,
  price: 99.99,
  stockEmpty: false,
  lastModified: ISODate("2024-01-11T11:36:17.956Z"),
  tags: [ 'fresh', 'green', 'natural', 'salad' ],
  productOwner: { name: 'john doe', source: 'bangalore', contact: '8726547861' },
  reviews: {

   '1': { _id: '1234', review: 'good taste' },
   '2': { _id: '1235', review: 'cheap and tasty' },
   title: 'users Reviews'
  },
  image: { name: Binary.createFromBase64("aGVsbG8gd29ybGQ=", 0) }
 }
]
e-com>
```

Geospatial Data Types in MongoDB

MongoDB provides specialized data types that enable applications to harness geospatial capabilities, facilitating the storage and retrieval of location-based information. Whether utilized for mapping, location-based services, or spatial analytics, MongoDB's geospatial features present a powerful and dependable solution. GeoJSON, a format representing geospatial data on a sphere similar to Earth, is key in enhancing these capabilities.

For example, we can add the "location" field as an array containing longitude and latitude values.

```
Unset
e-com> db.products.updateOne( { name: "tomato" }, { $set: {location:
[-122.4194, 37.7749]}} )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
e-com> db.products.find()
[
  {
    _id: ObjectId("659fc99c8022ec1ecd7a5524"),
    name: 'tomato',
    category: 'Vegetables',
    quantity: 20,
    price: 99.99,
    stockEmpty: false,
    lastModified: ISODate("2024-01-11T11:36:17.956Z"),
    tags: [ 'fresh', 'green', 'natural', 'salad' ],
    productOwner: { name: 'john doe', source: 'bangalore', contact:
'8726547861' },
    reviews: {
      '1': { _id: '1234', review: 'good taste' },
      '2': { _id: '1235', review: 'cheap and tasty' },
      title: 'users Reviews'
    },
    image: { name: Binary.createFromBase64("aGVsbG8gd29ybGQ=", 0) },
    location: [ -122.4194, 37.7749 ]
  }
]
e-com>
```

Min Key

In MongoDB, the MinKey is a special type used as a placeholder to represent the minimum possible value within an index. It is primarily used for range queries and comparisons in situations where you want to find documents with the smallest possible value.

For example, we can add minStock and represent with Minkey() i.e

```
Unset
e-com> db.products.updateOne( { name: "tomato" }, { $set: {minStock: MinKey()}
})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,


  upsertedCount: 0
}
e-com> db.products.findOne({name: "tomato"})
{
  _id: ObjectId("659fc99c8022ec1ecd7a5524"),
  name: 'tomato',
  category: 'Vegetables',
  quantity: 20,
  price: 99.99,
  stockEmpty: false,
  lastModified: ISODate("2024-01-11T11:36:17.956Z"),
  tags: [ 'fresh', 'green', 'natural', 'salad' ],
  productOwner: { name: 'john doe', source: 'bangalore', contact: '8726547861'
},
  reviews: {
    '1': { _id: '1234', review: 'good taste' },
    '2': { _id: '1235', review: 'cheap and tasty' },
    title: 'users Reviews'
  },
  image: { name: Binary.createFromBase64("aGVsbG8gd29ybGQ=", 0) },
  location: [ -122.4194, 37.7749 ],
  minStock: MinKey()
}
e-com>
```

Max Key

In MongoDB, the MaxKey is a special type used as a placeholder to represent the maximum possible value within an index. It is primarily used for range queries and comparisons in situations where you want to find documents with the largest possible value.

For example, we can add maxStock field to represent the maximum number of stock that can be stored. I.e.

```
Unset
e-com> db.products.updateOne( { name: "tomato" }, { $set: {maxStock: MaxKey()}
})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
e-com> db.products.findOne({name: "tomato"})
{
  _id: ObjectId("659fc99c8022ec1ecd7a5524"),
  name: 'tomato',
  category: 'Vegetables',
  quantity: 20,
  price: 99.99,
  stockEmpty: false,
  lastModified: ISODate("2024-01-11T11:36:17.956Z"),
  tags: [ 'fresh', 'green', 'natural', 'salad' ],
  productOwner: { name: 'john doe', source: 'bangalore', contact: '8726547861'
},
  reviews: {
    '1': { _id: '1234', review: 'good taste' },
    '2': { _id: '1235', review: 'cheap and tasty' },
    title: 'users Reviews'
  },
  image: { name: Binary.createFromBase64("aGVsbG8gd29ybGQ=", 0) },
  location: [ -122.4194, 37.7749 ],
  minStock: MinKey(),
  maxStock: MaxKey()
}
e-com>
```

Schema Design Considerations for Data Types

MongoDB, being a flexible NoSQL database, allows developers to design schemas that suit the specific needs of their applications. When designing the schema for your MongoDB database, careful consideration of data types is crucial to ensure optimal performance, efficient queries, and scalability. In this article, we will explore some essential considerations for choosing data types in MongoDB schema design.

1. Choose Appropriate Data Types

MongoDB supports a variety of data types, including strings, numbers, dates, arrays, and more. Choose data types that accurately represent the nature of your data.

For example, use Date for timestamp fields, String for textual data, and Number for numeric values.

```
Unset
{
  "name": "John Doe",
  "birthdate": ISODate("1990-01-01T00:00:00Z"),
  "scores": [95, 88, 72]
}
```

2. Consider Indexing

Data types affect how MongoDB indexes work. Choose appropriate data types for fields that need to be indexed for faster query performance.

For example, use ObjectId for primary keys and ensure that frequently queried fields are indexed.

```
Unset
{
  "_id": ObjectId("5f62a5ec89ee49391b56d525"),
  "username": "john_doe"
}
```

3. Leverage Embedded Documents and Arrays

MongoDB allows the nesting of documents and arrays within documents. Utilize embedded documents and arrays to represent relationships between data. This can reduce the need for multiple collections and enhance query performance.

For example, we can take an example of books and author

```
Unset
{
  "_id": 1,
  "author": {
    "name": "Jane Smith",
    "age": 35
  },
  "books": [
    {"title": "MongoDB Basics", "year": 2022},
    {"title": "Advanced MongoDB", "year": 2023}
  ]
}
```

4. Consider the Use of BSON Types

MongoDB employs BSON (Binary JSON) to represent data. Be aware of BSON types like ObjectId, Timestamp, and Binary and use them appropriately.

For example, ObjectId is commonly used as a unique identifier.

```
Unset
{
  "_id": ObjectId("507f1f77bcf86cd799439011"),
  "timestamp": Timestamp(1632300000, 1),
  "binaryData": new Binary(Buffer.from("..."))
}
```

5. Plan for Growth and Scalability

Consider the growth of your data and design the schema to scale efficiently. Avoid embedding large arrays within documents, as this can lead to document size limitations. Plan for sharding strategies to distribute data across multiple servers.

Handling Null and Undefined Values

Handling null and undefined values in MongoDB is an important aspect of database design, ensuring data consistency, query efficiency, and a smooth user experience. In this section, we will explore best practices for dealing with null and undefined values in MongoDB.

Undefined Values

In MongoDB, the term "undefined" refers to the absence of a field or key in a document. When a field is not present in a document, it is considered "undefined." This concept is distinct from a field having a value of null or any other specific data type; instead, it signifies the non-existence of the field.

For example -

```
Unset
// Document with an undefined field
{
  "_id": 1,
  "name": "John Doe"
  // The "age" field is undefined
}
```

From the above example, the document has a field named "name" with the value "John Doe," but it does not have a field named "age." The absence of the "age" field in the document means that it is considered undefined.

Null

In MongoDB, the null value represents the absence of a value or the explicit assignment of no value to a field within a document. It is a specific data type used to denote the intentional lack of data for a particular field. Unlike an undefined field, which means the field is not present in the document, a field with a value of null is explicitly set to represent the absence of a meaningful value.

Here's an example of a document with a field explicitly set to null:

```
Unset
{

   "_id": 1,
   "name": "John Doe",
   "age": null
 }
```

From the above example, the document has a field named "age" that is set to null. This can be useful when you want to distinguish between a field that has no meaningful value and a field that is simply not present in the document.

Data Type Conversion in MongoDB

Data type conversion plays a crucial role in database management, and MongoDB, being a flexible NoSQL database, offers various features for handling different data types. In this section, we'll explore best practices and considerations for data type conversion in MongoDB.

1. Understanding MongoDB Data Types

MongoDB supports a variety of data types, including strings, numbers, arrays, documents, dates, and more. Before diving into data type conversion, it's essential to have a solid understanding of the available data types and their representations in MongoDB.

For example

```
Unset
{
  "name": "John Doe",
  "age": 30,
  "scores": [95, 88, 72],
  "address": {
    "city": "Example City",
    "zip": "12345"
  },
  "isStudent": true,
  "registrationDate": ISODate("2022-01-01T00:00:00Z")
}
```

2. Implicit vs Explicit Conversion

MongoDB performs implicit data type conversion in certain situations, such as when comparing values in queries. However, it's generally a good practice to perform explicit conversions to ensure clarity and avoid unexpected behavior.

For example - Explicitly convert a string to an integer

```
Unset
db.collection.find({ "age": { $eq: NumberInt("30") } })
```

3. Numeric Conversion

When dealing with numeric values, MongoDB provides specific types like NumberInt and NumberLong for 32-bit and 64-bit integers, respectively. Explicitly using these types can help prevent unexpected rounding or loss of precision

For example - Explicitly convert a number to 32-bit integer

```
Unset
db.collection.updateOne({ "_id": 1 }, { $set: { "quantity": NumberInt(42) } })
```

4. Date Conversion

Dates are often represented using the ISODate constructor in MongoDB. When dealing with date-related operations, ensure that the dates are properly converted and formatted.

For example - Convert a string to ISODate

```
Unset
db.collection.find({ "registrationDate": ISODate("2022-01-01T00:00:00Z") })
```

5. Boolean Conversions

Boolean values are straightforward, but ensure consistency in representing true and false values. Use the Boolean constructor for explicit Boolean conversions.

For example, Convert a string to a boolean

```
Unset
db.collection.updateOne({ "_id": 1 }, { $set: { "isActive": Boolean("true") } })
```

Regular Expressions with String Data

Regular expressions (regex) provide a powerful tool for string manipulation and searching. MongoDB, being a versatile NoSQL database, supports the use of regular expressions for querying and manipulating string data. In this section, we'll explore how to harness the potential of regular expressions with string data in MongoDB.

Let's have a look at some of the uses of Regular Expressions with string data in MongoDB

1. Basics of Regular Expressions

A regular expression is a sequence of characters that defines a search pattern. MongoDB utilizes the PCRE (Perl Compatible Regular Expressions) syntax, allowing for flexible and powerful pattern matching.

For example - Find documents where the "name" field starts with "John"

```
Unset
db.collection.find({ "name": /^John/ })
```

2. Using the $regex operator

In MongoDB, the $regex operator enables the application of regular expressions in queries. This operator can be particularly useful when you need to search for documents with string fields matching a specific pattern.

For example - // Find documents where the "email" field contains "example.com"

```
Unset
db.users.find({ "email": { $regex: /example\.com/ } })
```

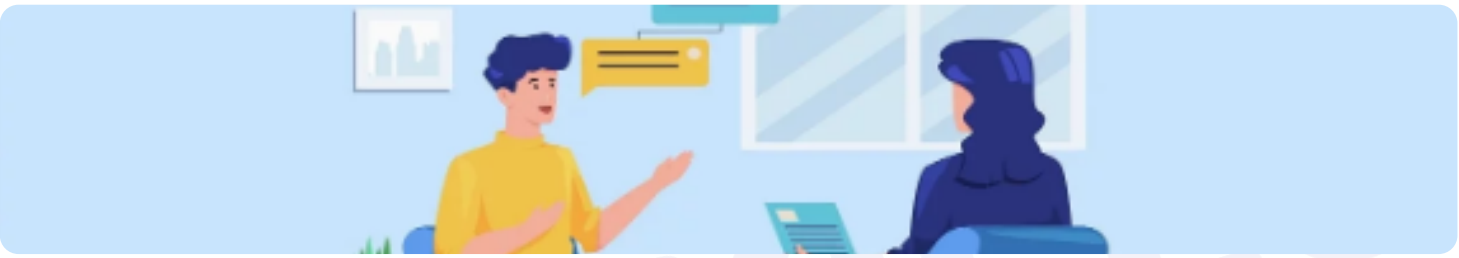
3. Case-Insensitive matching

Regular expressions in MongoDB queries are case-sensitive by default. To perform a case-insensitive search, use the $options modifier with the 'i' flag.

For example - Find documents where the "code" field is exactly "ABC123"

```
Unset
db.products.find({ "code": /^ABC123$/ })
```

Interview points



Question 1 - Explain the purpose of the ObjectId data type in MongoDB.

Answer - ObjectId is a 12-byte identifier typically employed as a unique identifier for documents. It consists of a timestamp, machine identifier, process identifier, and a random incrementing value.

Question 2 - Can you explain the basic data types supported by MongoDB?

Answer - MongoDB supports various data types, including String, Number, Boolean, Object, Array, Date, ObjectId, and more. Each data type is used to represent different kinds of values in a document.

PW SKILLS

THANK YOU !