Documents and Data Types

Objectives

- By the end of this session, you should be able to:
  - Introduction to JSON
  - BSON
  - MongoDB Documents
  - MongoDB Data Types
  - Limits and Restrictions on Documents
  - Field Name Rules
-

Introduction to JSON

- JSON is a full-text, lightweight format for data representation and transportation. JavaScript's simple representation of objects gave birth to JSON.
- JSON offers a human-readable, plain-text way of representing data. In comparison to XML, where information is wrapped inside tags, and lots of tags make
- it look bulky, JSON offers a compact and natural format where you can easily focus on the information.
- Different programming languages have different ways of representing language constructs, such as objects, lists, arrays, variables, and more. When two systems, written in two different programming languages, want to exchange data, they need to have a mutually agreed standard for representing information. JSON provides that standard with its lightweight format.

JSON Syntax

- JSON documents or objects are a plain-text set of zero or more key-value pairs. The key-value pairs form an object, and if the value is a collection of zero or more values, they form an array. JSON has a very simple structure where, by only using a set of curly braces ({}), square brackets ([]), colons (:), and commas(,), you can represent any complex piece of information in a compact form.
- In a JSON object, key-value pairs are enclosed within curly braces: {}. Within an object, the key is always a string. However, the value can be any of JSON's specified types.

```
{
        key : value
}
```

Example Of JSON data

- let's consider a sample JSON document that contains the basic information of a company.

```
{
"company_name" : "Sparter", "founded_year" : 2007, "twitter_username" : null,
"address" : "15 East Street", "no_of_employees" : 7890,
"revenue" : 879423000
}
```

- From the preceding document, we can see the following:
- Company name and address, both being string fields
- Foundation year, number of employees, and revenue as numeric fields
- The company's Twitter username as null or no information

JSON Data Types

- String: Refers to plain text
- • Number: Consists of all numeric fields
- • Boolean: Consists of True or False
- Introduction to JSON | 51
- Object: Other embedded JSON objects
- Array: Collection of fields
- Null: Special value to denote fields without any value {audience_rating: 6}
- {audience_rating: 7.6}

JSON Data Types

- As per the JSON specification, a number is just a sequence of digits. It does not differentiate between numbers such as integer, float, or long. Additionally, it restricts the range limits of numbers.

```
{audience_rating: 6}

{audience_rating: 7.6}
```

- JSON documents do not support the Date data type, and all dates are represented as plain strings.

.

Exercise 2.01: Creating Your Own JSON Document

- Your organization wants to build a dataset of movies and series, and they want to use MongoDB to store the records. As a proof of concept, they ask you to choose a random movie and represent it in JSON format.
  - 1. Open a JSON validator—for example, https://jsonlint.com/.
  - 2. Type the preceding information in JSON format
  - 3. Click on **`Validate JSON`** to validate the code.

BSON

- MongoDB documents are stored in a binary format called BSON. BSON documents are not human-readable, and you will never have to deal with them directly.
- BSON specifications are primarily based on JSON as they inherit all the good features of JSON, such as the syntax and flexibility.
- BSON documents are designed to be more efficient than JSON as they occupy
- less space and provide faster traversal.
- • With each document, BSON stores some meta-information, such as the length of the fields or the length of the sub-documents. The meta-information makes the document parsing, as well as traversing, faster.

MongoDB Documents

- A MongoDB database is composed of collections and documents. A database can have one or more collections, and each collection can store one or more related BSON documents. In comparison to RDBMS, collections are analogous to tables and documents are analogous to rows within a table. However, documents are much more flexible compared with the rows in a table. less space and provide faster traversal.

Features of MongoDB's document-based data model

- 1. The documents provide a flexible and natural way of representing data. The data
- can be stored as is, without having to transform it into a database structure.
- 2. The objects, nested objects, and arrays that are within a document are easily relatable to your programming language's object structure.
- 3. With the ability of a flexible schema, the documents are agile in practice. They continuously integrate with application changes and new features without any major schema changes or downtimes.
- 4. Documents are self-contained pieces of data. They avoid the need to read multiple relational tables and table-joins to understand a complete unit of information.
- 5. The documents are extensible. You can use documents to store the entire object structure, use it as a map or a dictionary, as a key-value pair for quick lookup, or have a flat structure that resembles a relational table.

MongoDB Data Types

- Strings

A string is a basic data type used to represent text-based fields in a document. It is a plain sequence of characters. can be stored as is, without having to transform it into a database structure.

```
{
   "name" : "Tom Walter"
}
```

- **Numbers**

A number is JSON's basic data type. A JSON document does not specify whether a number is an integer, a float, or long

```
{
"number_of_employees": 50342
}
```

MongoDB Data Types

- MongoDB supports the following types of numbers:
- • **`double`**: 64-bit floating point
- • int: 32-bit signed integer
- • **`long`**: 64-bit unsigned integer
- • **`decimal`**: 128-bit floating point – which is IEE 754-compliant
- If you are working on the mongo shell, you get three wrappers to handle: **`integer`**, **`long`**, and **`decimal`**. The Mongo shell is based on JavaScript, and thus all the documents are represented in JSON format. By default, it treats any number as a
- 64- bit floating point. However, if you want to explicitly use the other types, you can use the following wrappers.
- **`NumberInt`**: The **`NumberInt`** constructor can be used if you want the number to be saved as a 32-bit integer and not as a 64-bit float:

- `var plainNum = 1299`
- `> var explicitInt = NumberInt("1299")`

MongoDB Data Types

- If you are working on the mongo shell, you get three wrappers to handle: **`integer`**, **`long`**, and **`decimal`**. The Mongo shell is based on JavaScript, and thus all the documents are represented in JSON format. By default, it treats any number as a 64- bit floating point. However, if you want to explicitly use the other types, you can use the following wrappers.
- **`NumberInt`**: The **`NumberInt`** constructor can be used if you want the number to be saved as a 32-bit integer and not as a 64-bit float:

```
> var plainNum = 1299

> var explicitInt = NumberInt("1299")

> var explicitInt_double = NumberInt(1299)
```

- In the preceding snippet, the first number, **`plainNum`**, is initialized with a sequence of digits without mentioning any explicit data type. Therefore, by default, it will be treated as a *64-bit floating-point number* (also known as a double).

MongoDB Data Types

- **`NumberDecimal`**: This wrapper stores the given number as a 128-bit IEEE 754 decimal format. The `NumberDecimal` constructor accepts both a string and a double representationn of the number:

```
var explicitDecimal = NumberDecimal("142.42")

> var explicitDecimal_double = NumberDecimal(142.42)
```

- **Booleans** :The Boolean data type is used to represent whether something is true or false. Therefore, the value of a valid Boolean field is either **`true`** or **`false`** .
- **Objects** The object fields are used to represent nested or embedded documents—that is, a field whose value is another valid JSON document.
- **Arrays** :A field with an **array** type has a collection of zero or more values. In MongoDB, there is no limit to how many elements an array can contain or how many arrays a document can have. However, the overall document size should not exceed 16 MB.

MongoDB Data Types

- **Arrays example:**

  - `> var doc = { first_array: [ 4, 3, 2, 1 ] }`

    - Each element in an array can be accessed using its index position. While accessing an element on a specific index position, the index number is enclosed in square brackets. Let's print the third element in the array:

      - `> doc.first_array[3]`
      - `Output :1`

MongoDB Data Types

- **Arrays example:**

  - `> var doc = { first_array: [ 4, 3, 2, 1 ] }`

    - Each element in an array can be accessed using its index position. While accessing an element on a specific index position, the index number is enclosed in square brackets. Let's print the third element in the array:

      If you print the array, you will see the embedded array as follows:

      - `> doc.first_array[3] Output :1`

- Just like objects having embedded objects, arrays can also have embedded arrays. The following syntax adds an embedded array into the sixth element:
- `doc.first_array[5] = [11, 12]`
- If you print the array, you will see the embedded array as follows:

  - `doc.first_array`

    - `[ 4, 3, 2, 1, 99, [11, 12]]`

MongoDB Data Types

- The array can contain any MongoDB valid data type fields. This can be seen in the following snippet:

```
// array of strings
[ "this", "is", "a", "text" ]
// array of doubles [ 1.1, 3.2, 553.54 ]
// array of Json objects
[ { "a" : 1 }, { "a" : 2, "b" : 3 }, { "c" : 1 } ]
// array of mixed elements
[ 12, "text", 4.35, [ 3, 2 ], { "type" : "object" } ]
```

MongoDB Data Types

- The array can contain any MongoDB valid data type fields. This can be seen in the following snippet:
- `// array of strings`
- `[ "this", "is", "a", "text" ]`
- `// array of doubles [ 1.1, 3.2, 553.54 ]`
- `// array of Json objects`
- `[ { "a" : 1 }, { "a" : 2, "b" : 3 }, { "c" : 1 } ]`
- `// array of mixed elements`
- `[ 12, "text", 4.35, [ 3, 2 ], { "type" : "object" } ]`

If you print the array, you will see the embedded array as follows:

Exercise 2.03: Using Array Fields

- In order to add comment details for each movie, your organization wants you to include full text of the comment along with user details such as name, email, and date. Your task is to prepare two dummy comments and add them to the existing movie record.

- validate the JSON document with an online validator

If you print the array, you will see the embedded array as follows:

MongoDB Data Types

- **Null** is a special data type in a document and denotes a field that does not contain a value. The **`null`** field can have only **`null`** as the value. You will print the object in the following example, which will result in the **`null`** value:

```
var obj = null >

> obj Null
```

- **ObjectId** Every document in a collection must have an **`_id`** that contains a unique value. This field acts as a *primary key* to these documents. The primary keys are used to uniquely identify the documents, and they are always indexed. The value of the **`_id`** field must be unique in a collection. When you work with any dataset, each dataset represents a different context, and based on the context, you can identify whether your data has a primary key.
- If you insert a document without an **`_id`** field, the MongoDB driver will autogenerate a unique ID and add it to the document. So, when you retrieve the inserted document, you will find **`_id`** is generated with a unique value of random text. When the **`_id`** field is automatically added by the driver, the value is generated using ObjectId. The **`ObjectId`** value is designed to generate lightweight code that is unique across different machines. It generates a unique value of 12 bytes, where the first 4 bytes represent the timestamp, bytes 5 to 9 represent a random value, and the last 3 bytes are an incremental counter.

MongoDB Data Types

- Create and print an **`ObjectId`** value as follows:

- `> var uniqueID = new ObjectId()`

Print **`uniqueID`** on the next line:

- `> uniqueID ObjectId("5dv.8ff48dd98e621357bd50")`

- MongoDB supports a technique called sharding, where a dataset is distributed and stored on different machines. When a collection is sharded, its documents are physically located on different machines. Even so, **`ObjectId`** can ensure that the values will be unique in the collection across different machines. If the collection is sorted using the **`ObjectId`** field, the order will be based on the document creation time. However, the timestamp in **`ObjectId`** is based on the number of seconds to epoch time. Hence, documents inserted within the same second may appear in a random order. The **`getTimestamp()`** method on **`ObjectId`** tells us the document insertion time.

MongoDB Data Types

- **Dates**

The JSON specifications do not support date types. All the dates in JSON documents are represented as plain strings. The string representations of dates are difficult to parse, compare, and manipulate. MongoDB's BSON format, however, supports **Date** types explicitly.

The MongoDB dates are stored in the form of milliseconds since the Unix epoch, which is January 1, 1970. To store the millisecond's representation of a date, MongoDB uses a 64-bit integer (**`long`**). Because of this, the date fields have a range of around +/-290 million years since the Unix epoch. One thing to note is that all dates are stored in *UTC*, and there is no *time zone* associated with them.

While working on the mongo shell, you can create **`Date`** instances using **`Date()`**, **`new Date()`**, or **`new ISODate()`**:

- `var date = Date()`

```
// Sample output

Sat Sept 03 1989 07:28:46 GMT-0500 (CDT)
```

When a **`Date()`** type is used to construct a date, it uses JavaScript's date representation, which is in the form of plain strings.

MongoDB Data Types

- **Dates :**

If you add the **`new`** keyword to the **`Date`** constructor, you get the BSON date that is wrapped in **`ISODate()`** as follows:

```
> var date = new Date()

// Sample output

ISODate("1989-09-03T10:11:23.357Z")
```

- You can also use the **`ISODate()`** constructor directly to create **`date`** objects as follows:

```
var isoDate = new ISODate()

// Sample output

ISODate("1989-09-03T11:13:26.442Z")
```

MongoDB Data Types

- **Timestamps**

The timestamp is a 64-bit representation of date and time. Out of the 64 bits, the first 32 bits store the number of seconds since the Unix epoch time, which is January 1, 1970. The other 32 bits indicate an incrementing counter. The timestamp type is exclusively used by MongoDB for internal operations.

- **Binary Data** also called **`BinData`**, is a BSON data type for storing data that exists in a binary format. This data type gives you the ability to store almost anything in the database, including files such as text, videos, music, and more.

```
{
"name" : "my_txt",
"extension" : "txt", "content" : BinData(0,/
"VGhpcyBpcyBhIHNpbXBsZSB0ZXh0IGZpbGUu")
}
```

Limits and Restrictions on Documents

- MongoDB has put some limits and restrictions on documents. One thing to note is that the restrictions are not because of database limitations or shortcomings. The restrictions are added so that the overall database platform can perform efficiently.
- **Document Size Limit** A document with too much information is bad in many ways. For this reason, MongoDB puts a limit of 16 MB on the size of every document in the collection. The limit of 16 MB is enough to store the right information. A collection can have as many documents as you want. There is no limitation on the size of a collection.
- **Nesting Depth Limit** A MongoDB BSON document supports nesting up to 100 levels, which is more than enough. Nested documents are a great way to provide readable data.

By setting the nesting limit of 100 levels, MongoDB avoids such issues. However, if you can't avoid such deep nesting, you can consider splitting the collections into two, or more, and have document references.

Limits and Restrictions on Documents

- **Field Name Rules**

1. The field name cannot contain a **null** character.

2. Only the fields in an array or an embedded document can have a name starting with the dollar sign (**$**). For the top-level fields, the name cannot start with a dollar (**$**) sign.

3. Documents with duplicate field names are not supported. According to the MongoDB documentation, when a document with duplicate field names is inserted, no error will be thrown, but the document won't be inserted. Even the drivers will drop the documents silently. On the mongo shell, however, if such a document is inserted, it gets inserted correctly.

- **Note**

MongoDB (as of version 4.2.8) does not recommend field names starting with a dollar (**$**) sign or a dot (**.**). The MongoDB query language may not work correctly with such fields. Additionally, the drivers do not support them.

Limits and Restrictions on Documents

- **Field Name Rules**

1. The field name cannot contain a **null** character.

2. Only the fields in an array or an embedded document can have a name starting with the dollar sign (**$**). For the top-level fields, the name cannot start with a dollar (**$**) sign.

3. Documents with duplicate field names are not supported. According to the MongoDB documentation, when a document with duplicate field names is inserted, no error will be thrown, but the document won't be inserted. Even the drivers will drop the documents silently. On the mongo shell, however, if such a document is inserted, it gets inserted correctly.

- **Note**

MongoDB (as of version 4.2.8) does not recommend field names starting with a dollar (**$**) sign or a dot (**.**). The MongoDB query language may not work correctly with such fields. Additionally, the drivers do not support them.

Exercise 2.04: Loading Data into an Atlas Cluster

- **Follow the steps**

Summary

- In this chapter, we have covered a detailed structure of MongoDB documents and document-based models, which is important before we dive into more advanced concepts in the upcoming chapters. We began our discussion with the transportation and storage of information in the form of JSON-like documents that provide a flexible and language-independent format. We studied an overview of JSON documents, the document structure, and basic data types, followed by BSON document specifications and differentiating between BSON and JSON on various parameters.