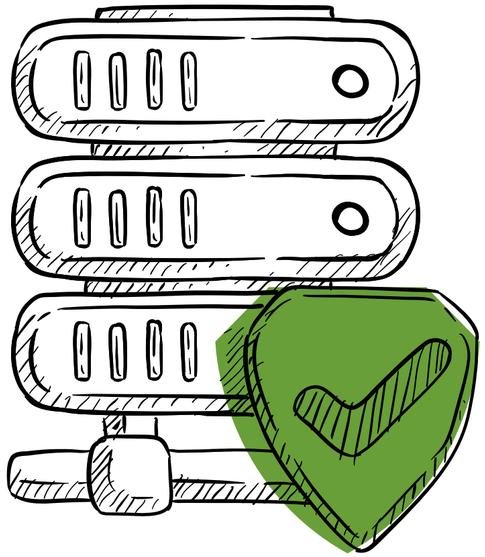
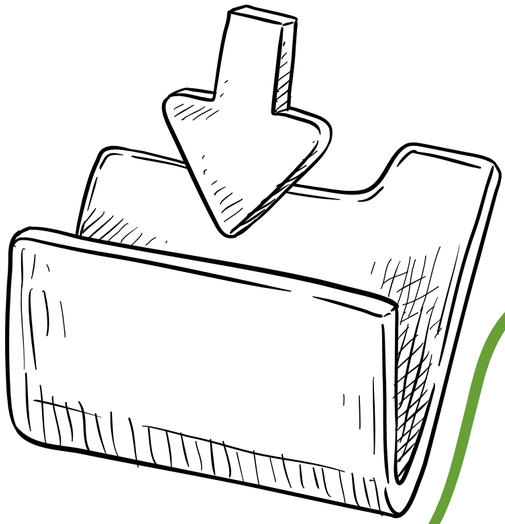
MONGO DATABASE

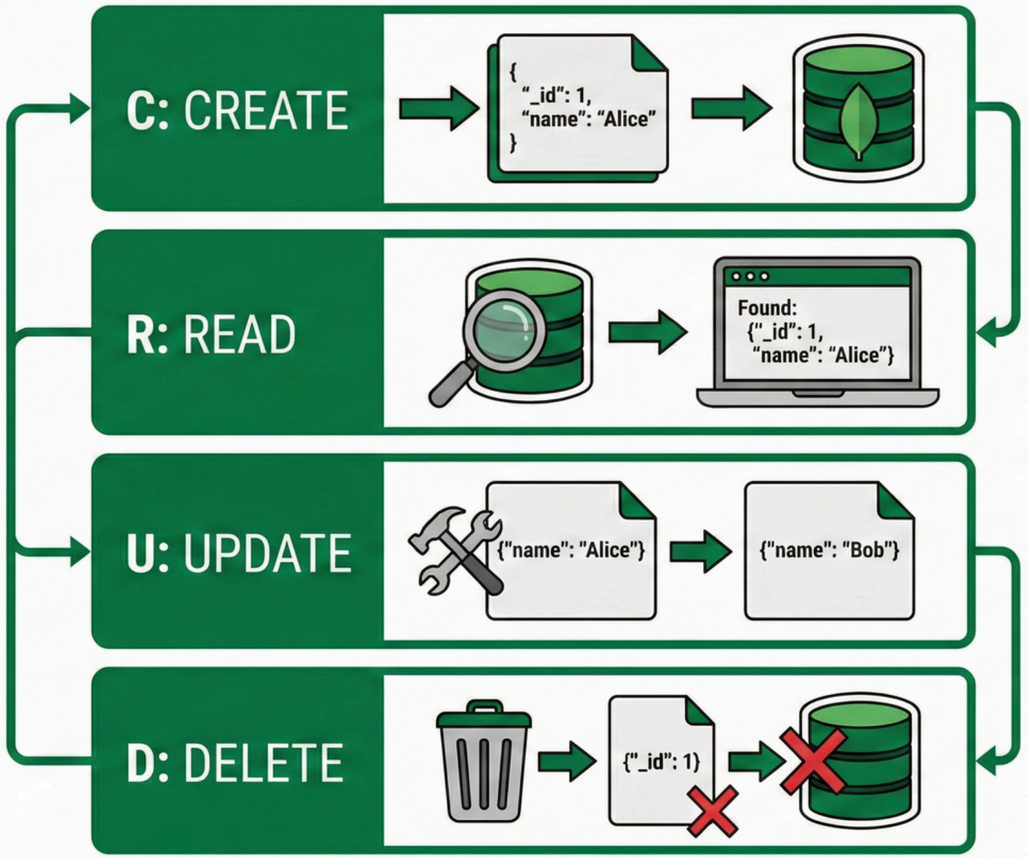
DAY02

Prepared by
Noha Shehab





CRUD Operations in MongoDB



INSERT

- insertOne(): insert only one document into a collection
- insertMany(): insert multiple documents into a collection

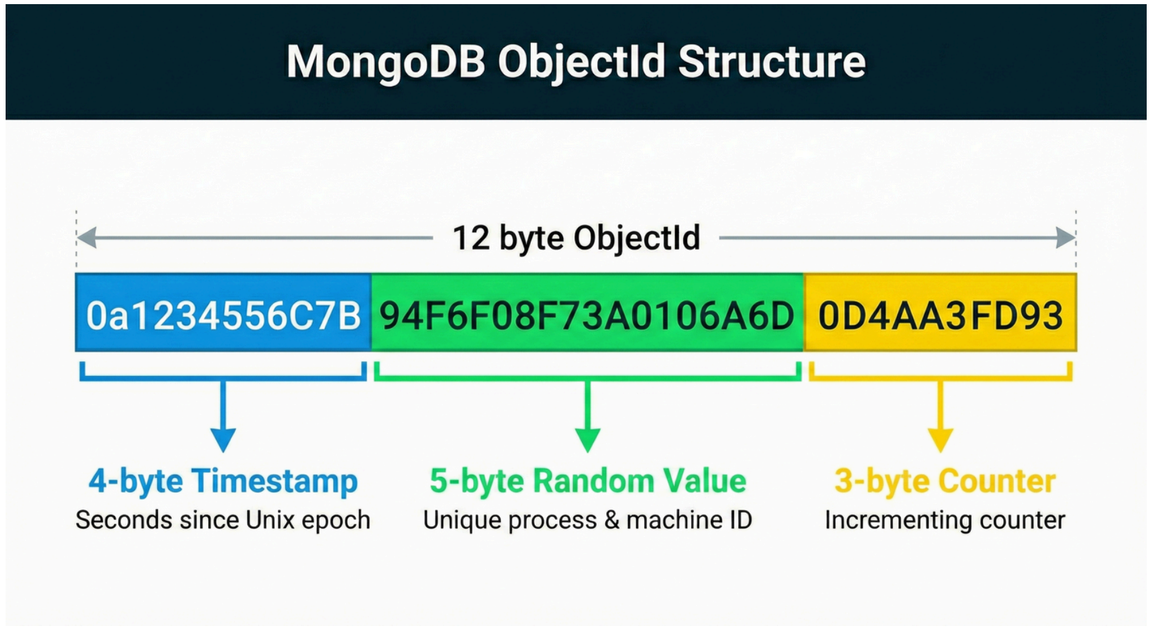
commands

```
db.users.insertOne({
  name: "Noha Shehab",
  email: "nshehab@iti.gov.eg",
  age: 33,
  status: "active"
});

db.products.insertMany([
  { item: "Laptop", qty: 25, tags: ["electronics", "office"] },
  { item: "Coffee Mug", qty: 100, tags: ["kitchen", "home"] },
  { item: "Desk Chair", qty: 15, tags: ["furniture"] }
]);
```

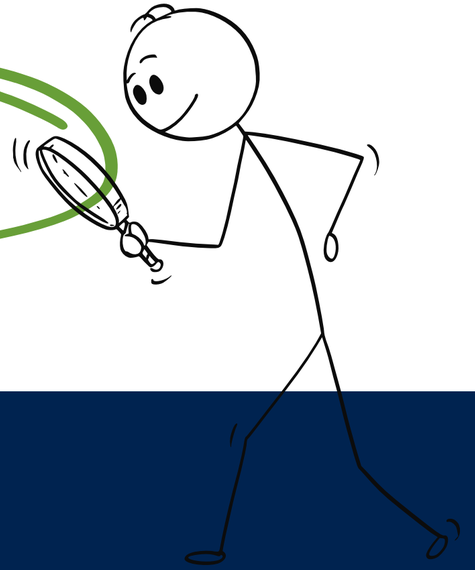
OBJECT ID

ObjectId is the default type for the _id field in every document.

- It is a **12-byte** identifier that is designed to be unique, fast to generate, and ordered by time.
- Think of it as a more advanced version of an auto-incrementing integer (like you'd find in SQL), but decentralized—meaning multiple servers can generate them simultaneously without accidentally creating the same ID.



FIND..



commands

```
Find All     ⟶ db.users.find({})
Exact Match  ⟶ db.users.find({ "city": "London" })
Comparison   ⟶ db.users.find({ "age": { "$gte": 21 } }) (Age ≥ 21)
Sort & Limit⟶ db.users.find().sort({ "joinDate": -1 }).limit(10)
```

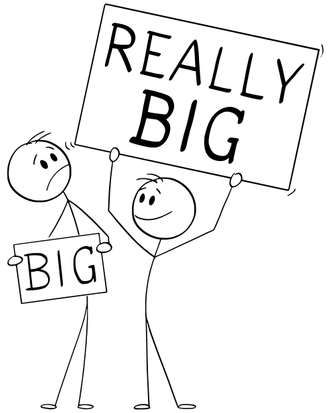
commands

```
Projection

db.instructors.find(
    {}, // condition
    {firstName:1 , lastName:1} // projection
)
```

FIND OPERATORS

- **Find operators**
  - Comparison Operators.
  - Logical Operators.
  - Element operators
  - Array operators
  - Evaluation Operators

COMAPARISON



Operator	Description	Example Scenario
$eq	Matches values that are equal to a specified value. (Often implicit).	Find users whose status is explicitly "active".
$ne	Matches all values that are **not equal** to a specified value.	Find products that are NOT colored "red".
$gt	Matches values **greater than** a specified value.	Find prices over $100.
$gte	Matches values **greater than or equal** to a specified value.	Find users age 18 or older.
$lte	Matches values **less than or equal** to a specified value.	Find orders placed before or on a specific date.
$nin*	Matches none of the values specified in an array.	Find documents where the category is NOT "electronics" or "books".

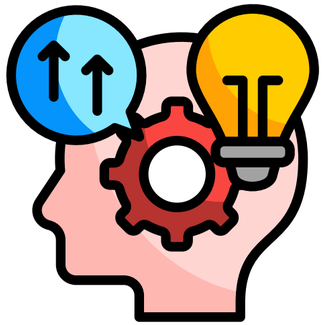
find.js

```
// Find products cheaper than 20
db.products.find({ price: { $lt: 20 } })

// Find users aged between 25 and 35 (implicit AND)
db.users.find({ age: { $gte: 25, $lte: 35 } })

// Find statuses that match a list
db.orders.find({ status: { $in: ["shipped", "delivered"] } })
```

LOGICAL

find.js

```
// The $or operator must take an array of objects
db.users.find({
  $or: [
    { status: "premium" },
    { age: { $gt: 60 } }
  ]
})
```



Operator	Description	Example Scenario
$or	Joins query clauses with a logical OR returns documents matching *any* of the clauses.	Find users who are either "admin" OR over age 50.
$and	Joins query clauses with a logical AND returns documents matching *all* clauses. (Often implicit with commas).	Find users who are "active" AND aged 25.
$not	Inverts the effect of a query expression.	Find documents where the price is NOT greater than 100 (similar to $lte).
$nor	Joins query clauses with a logical NOR returns documents that fail *all* clauses.	Find items that are neither "red" nor "blue".

ARRAY OPERTATORS

Operator	Description	Example Scenario
$all	Matches arrays that contain *all* the specified elements.	Find blog posts that have BOTH tags "coding" AND "tutorial".
$elemMatch	Matches documents that contain an array field with at least one element that matches all the specified criteria.	Find orders where at least one item in the items array is both "blue" AND costs over $10.
$size	Matches any array with the specified number of elements.	Find blog posts that have exactly 3 comments.

find.js

```
// Find posts tagged with both "mongodb" AND "database"
db.posts.find({ tags: { $all: ["mongodb", "database"] } })

// Assume a 'scores' array: [ { subject: "math", score: 80 },
 { subject: "science", score: 95 } ]
// Find students who have at least one subject with a score > 90
db.students.find({
  scores: {
    $elemMatch: { score: { $gt: 90 } }
  }
})
```

ELEMENT

Operator	Description	Example Scenario
$exists	Matches documents that have the specified field (true or false).	Find users who have a phone number field recorded, regardless of value.
$type	Matches documents where the value of a field is a specific BSON type.	Find documents where "zipcode" was accidentally saved as a string instead of a number.

find.js

```
// Find documents that HAVE the 'deletedAt' field
db.users.find({ deletedAt: { $exists: true } })

// Find documents where 'deletedAt' does NOT exist (active users)
db.users.find({ deletedAt: { $exists: false } })

// Find documents where 'age' is a string type (Type 2 refers to String)
db.users.find({ age: { $type: 2 } })
// Or use the alias
db.users.find({ age: { $type: "string" } })
```

EVALUATION OPERATORS

Operator	Description
$regex	Provides regular expression capabilities for pattern matching in strings.
$expr	Allows the use of aggregation expressions within the query language. Highly useful for comparing two different fields *within the same document*.
$mod	Performs a modulo operation on the value of a field and selects documents with a specified remainder.

find.js

```
// Find usernames that start with "john" (case insensitive)
db.users.find({ username: { $regex: /^john/, $options: 'i' } })

// Find documents where the 'spent' field is greater than the 'budget' field
db.budget.find({ $expr: { $gt: [ "$spent", "$budget" ] } })
```

PROJECTION

In MongoDB, projection is the process of determining which fields should be returned in the matching documents.

- If the first argument of **db.collection.find(query, projection)** is the "WHERE" clause of SQL, the second argument (the projection) is the "SELECT" clause.
- By default, MongoDB returns every field in a document. You use projection to limit this.

Why use Projection?

1. **Performance (Bandwidth)**: Sending smaller documents over the network is faster and cheaper.
2. **Memory usage:** The application server needs less RAM to hold the results.
3. **Security**: Preventing sensitive data (hashed passwords, internal flags, PII) from accidentally being sent to the client.

find.js

```
db.users.find(
    { status: "active" }, // The Query
    { name: 1, email: 1 } // The Projection
)
```

UPDATE

- **In a relational database (SQL),** updating a row is usually straightforward: UPDATE table SET column = value.
- In MongoDB, because documents have complex structures (nested objects and arrays), you need special update operators to tell the database exactly **how to modify a document** without overwriting the whole thing.
- All update operators start with a dollar sign **($).**
- They are used as the second argument in update commands like **updateOne()**, **updateMany()**, or **findOneAndUpdate().**

update.js

```
// This might replace the WHOLE document with just
//{ status: "inactive" }
db.users.updateOne({ _id: 1 }, { status: "inactive" })

// This only changes the status field, leaving the rest untouched.
db.users.updateOne({ _id: 1 }, { $set: { status: "inactive" } })
```

UPDATE OPERATORS

- Field operators
- Array operators
- The Positional Operator ($)

update.js

```
//The Golden Rule of Updates
//If you don't use an operator, you risk replacing the entire document.

// This might replace the WHOLE document with just { status: "inactive" }
db.users.updateOne({ _id: 1 }, { status: "inactive" })


// This only changes the status field, leaving the rest untouched.
db.users.updateOne({ _id: 1 }, { $set: { status: "inactive" } })
```

FIELD OPERATORS

Operator	Description	Example Scenario
$set	Sets the value of a field. If the field doesn't exist, it creates it. This is the most commonly used operator.	Changing a user's email address or adding a new "preferences" object.
$unset	Deletes a field completely from a document.	Removing a temporary "resetToken" field after it has been used.
$inc	Increments (or decrements) a numeric field by a specified amount. It is atomic and thread-safe.	Increasing a "viewCount" by 1, or decreasing inventory by 5.
$min	Updates the field only if the specified value is **less than** the current value.	Tracking a "lowestPriceSeen" for a product.
$max	Updates the field only if the specified value is **greater than** the current value.	Tracking a player's "highScore".
$rename	Renames a field.	Fixing a typo in a field name across all documents (e.g., changing "addres" to "address").
$currentDate	Sets the value of a field to the current date and time.	Updating a "lastModified" timestamp.

FIELD OPERATORS

update.js

```
// Scenario: A user bought an item.
db.users.updateOne(
   { _id: 123 },
   {
     $set: { status: "active", lastPurchaseDate: new Date() }, // Change/Add fields
     $inc: { points: 100, walletBalance: -19.99 },             // Math operations
     $unset: { cartId: "" }                                    // Remove field
   }
)

// Scenario: Update high score only if the new score is better
db.players.updateOne(
   { _id: 555 },
   { $max: { highScore: 9500 } } // If current is 9000, it updates. If 10000, it does nothing.
)
```

ARRAY OPERATORS

- Adding to Arrays
- Removing from Arrays

Operator	Description	Example
$push	Appends a value to an array. It will add duplicates if you run it twice.	Adding a new log entry to a "loginHistory" array.
$addToSet	Adds a value to an array **only if it doesn't already exist**. Ensures uniqueness within the array.	Adding a "tag" to a post (you don't want duplicate tags).

update.js

```
// Add multiple tags, ensuring no duplicates exist
db.posts.updateOne(
  { _id: 1 },
  { $addToSet: { tags: { $each: ["mongodb", "database", "nosql"] } } }
)
```

Operator	Description	Example
$pop	Removes the first (-1) or last (1) element of an array.	Removing the oldest entry from a capped history log.
$pull	Removes *all* array instances that match a specified condition. Very powerful.	Removing a specific comment by its ID, or removing all scores below 50.
$pullAll	Removes all instances of the specific values listed.	Removing both "red" and "blue" from a colors array.

update.js

```
// Remove the specific comment with id "c_555" from the 'comments' array
db.posts.updateOne(
  { _id: 1 },
  { $pull: { comments: { commentId: "c_555" } } }
)

// Remove all scores less than 60 from the 'scores' array
db.students.updateOne(
  { _id: 1 },
  { $pull: { scores: { $lt: 60 } } }
)
```

SCHEMA VALIDATION

Schema validation lets you create validation rules for your fields, such as allowed data types and value ranges.
MongoDB uses a **flexible schema model**, which means that documents in a collection do not need to have the same fields or data types by default.
Once you've established an application schema, you can use schema validation to ensure there are no unintended schema changes or improper data types.

Validation Levels

- strict (Default): MongoDB applies validation rules to all inserts and updates.
- moderate: Rules are applied to all inserts. For updates, rules are only applied to existing documents that already fulfilled the criteria.

Validation Actions

- error (Default): Rejects any document that violates the rules.
- warn: Allows the document to be saved but logs the violation in the MongoDB logs.

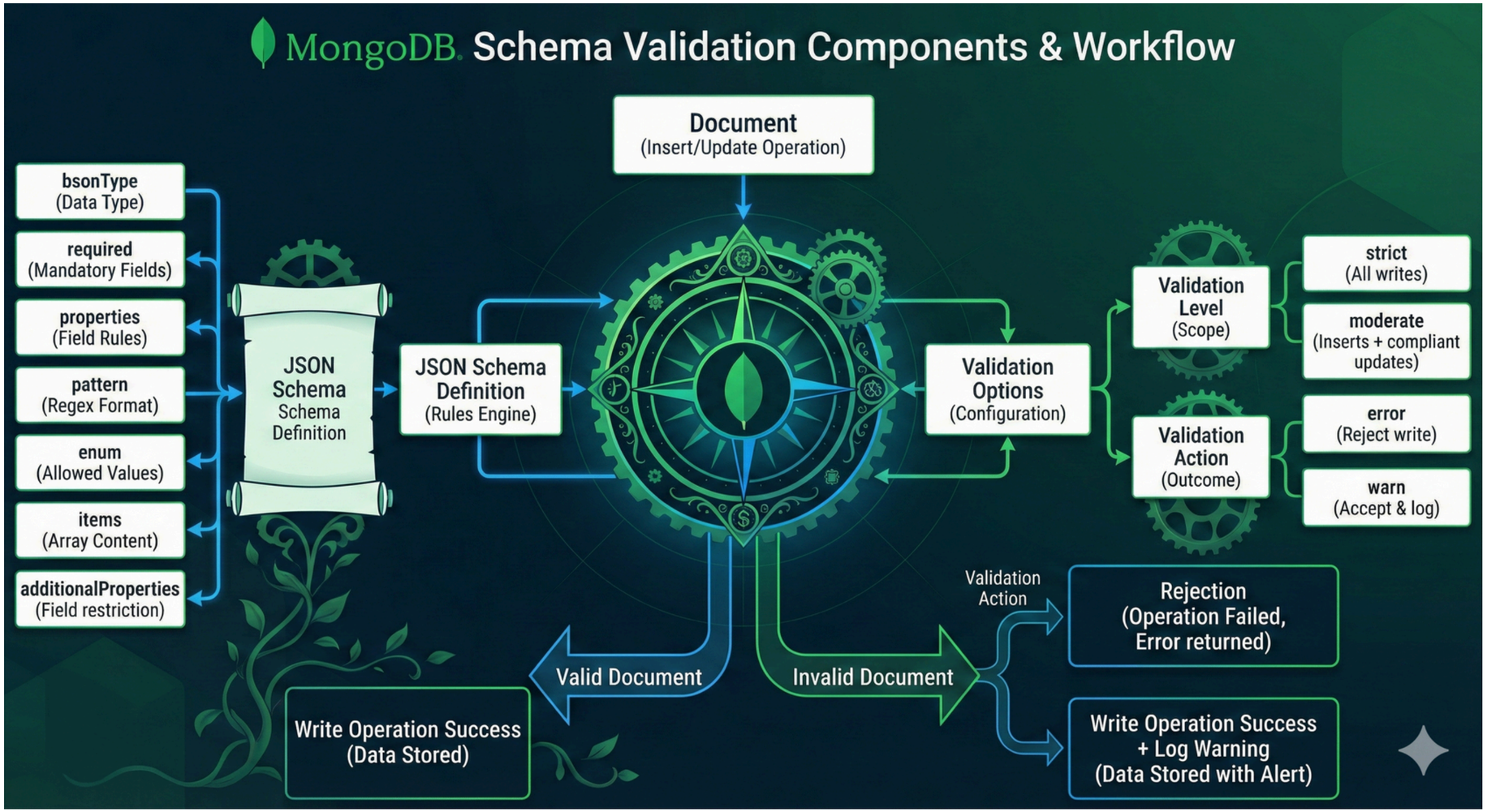
SCHEMA VALIDATION

- **You can add validation**
  - while creating schemas
  - to existing schemas

Key Validation Keywords

Keyword	Purpose
bsonType	Defines the data type (e.g., string, int, double, bool, objectId).
required	An array of fields that *must* be present in the document.
enum	Restricts a field to a fixed set of values (e.g., ['admin', 'editor', 'user']).
pattern	Uses Regex to validate the format of a string (like emails or phone numbers).
items	Validates the contents of an array.
additionalProperties	Set to false to prevent users from adding fields not defined in the schema.

VALIDATION PROCESS



CHECK THIS

schemaValidation.js

```
db.createCollection("users", {
   validator: {
      $jsonSchema: {
         bsonType: "object",
         required: ["name", "age"],
         properties: {
            name: {
               bsonType: "string",
               description: "must be a string and is required"
            },
            age: {
               bsonType: "int",
               minimum: 0,
               maximum: 120,
               description: "must be an integer in [ 0, 120 ] and is required"
            },
            email: {
               bsonType: "string",
               pattern: "^.+@.+$",
               description: "must be a string and match the regular expression pattern"
            }
         }
      }
   }
})
```

schemaValidation.js

```
db.runCommand({
   collMod: "users",
   validator: {
      $jsonSchema: {
         bsonType: "object",
         required: ["name", "age"],
         properties: {
            name: {
               bsonType: "string",
               description: "must be a string and is required"
            },
            age: {
               bsonType: "int",
               minimum: 0,
               maximum: 120,
               description: "must be an integer in [ 0, 120 ] and is required"
            },
            email: {
               bsonType: "string",
               pattern: "^.+@.+$",
               description: "must be a string and match the regular expression pattern"
            }
         }
      }
   },
   validationLevel: "moderate",
   validationAction: "error"
})
```