CRUD Operations: insertOne, find, updateOne, deleteOne.

1. Problem Statement

Case Study: FastBite Food Delivery

FastBite is building a scalable backend for managing restaurant menus and customer orders:

- New dishes are added and removed daily as menus change.
- Customers need to search for dishes by cuisine, price, or dietary tags.
- Prices and availability change frequently—updates must be fast and reliable.
- Orders and menu items must be deleted or archived when restaurants close or dishes are discontinued.

The challenge:
How do you ensure the system can efficiently create, read, update, and delete menu and order data, while maintaining data integrity and performance as the business grows?

---

2. Learning Objectives

By the end of this tutorial, you will:

- Understand the role and guarantees of CRUD operations in MongoDB.
- Use `insertOne` to add new documents to a collection.
- Use `find` with advanced queries, projections, and sorting to retrieve documents.
- Use `updateOne` with operators to modify documents safely and atomically.

- Use `deleteOne` to remove documents, understanding the importance of filters and atomicity.
- Apply best practices for data modeling, error handling, and performance.

---

3. Concept Introduction with Analogy

Analogy: The Restaurant Order Board

- **insertOne**: Like a chef pinning a new dish to the kitchen's order board.
- **find**: Scanning the board for all "Vegan" or "Under $10" dishes.
- **updateOne**: The manager changes the price or marks a dish as "out of stock."
- **deleteOne**: Removing a discontinued dish from the board.

MongoDB's CRUD operations are your digital tools for managing this ever-changing board—instantly, reliably, and at scale.

4. Technical Deep Dive

A. MongoDB CRUD Basics

- **Atomicity**: Each operation is atomic at the document level.
- **Document Structure**: Flexible schema allows nested fields (e.g., `price.currency`, `price.value`).
- **Indexes**: Improve query performance (e.g., indexing `category` or `price`).

B. `insertOne`: Adding Products

Purpose: Add a single document to a collection.

Syntax:

```
db.products.insertOne({
  name: "Wireless Headphones",
  category: "Electronics",
  price: { value: 99.99, currency: "USD" },
  stock: 50,
  ratings: [4.5, 4.7, 4.8]
});
```

Key Notes:

- MongoDB auto-generates a unique `_id` (primary key) if not provided.
- Returns an `acknowledged: true` and the inserted `_id`.

C. `find`: Querying Products

Purpose: Retrieve documents matching criteria.

Syntax:

```
// Find all electronics in stock
db.products.find({
  category: "Electronics",
  stock: { $gt: 0 }
});

// Projection: Include only name and price
db.products.find(
  { category: "Electronics" },
  { name: 1, price: 1, _id: 0 }
);

// Sorting: Highest price first
db.products.find().sort({ "price.value": -1 });
```

Operators:

- **Comparison**: `$eq`, `$gt`, `$lt`, `$in`.
- **Logical**: `$and`, `$or`, `$not`.
- **Array**: `$elemMatch`, `$size`.

D. `updateOne`: Modifying Products

Purpose: Update a single document matching a filter.

Syntax:

```
// Update stock quantity for a product
db.products.updateOne(
  { _id: ObjectId("...") },
  { $set: { stock: 45 } }
);

// Increment stock by 10
db.products.updateOne(
  { name: "Wireless Headphones" },
  { $inc: { stock: 10 } }
);

// Add a new field (e.g., discount)
db.products.updateOne(
  { _id: ObjectId("...") },
  { $set: { "price.discount": 15 } }
);
```

Key Notes:

- Use `$set`, `$inc`, `$push` (for arrays) to avoid overwriting the entire document.
- Always include a filter to prevent accidental updates.

E. `deleteOne`: Removing Products

Purpose: Delete a single document matching a filter.

Syntax:

```
// Delete a discontinued product
db.products.deleteOne({
  _id: ObjectId("...")
});

// Delete by name (ensure uniqueness first)
db.products.deleteOne({
  name: "Outdated Model XYZ"
});
```

Key Notes:

- Use precise filters to avoid accidental deletions.
- Consider archiving instead of deleting for historical data.

---

5. Step-by-Step Code Walkthrough

A. Define the Product Schema

// Sample product document

```
{
  _id: ObjectId("665f4d7e8b3e6c1e24a7b3e1"),
  name: "Smartwatch Pro",
  category: "Wearables",
  price: { value: 199.99, currency: "USD", discount: 20 },
  stock: 25,
  ratings: [4.6, 4.8],
  tags: ["fitness", "bluetooth"]
}
```

B. Insert a New Product

```
db.products.insertOne({
  name: "4K Smart TV",
  category: "Electronics",
  price: { value: 599.99, currency: "USD" },
  stock: 10,
  tags: ["television", "streaming"]
});
```

Output:

```
{
  acknowledged: true,
  insertedId: ObjectId("665f4d7e8b3e6c1e24a7b3e2")
}
```

C. Query Products

```
// Find all TVs under $600
db.products.find({
  category: "Electronics",
  "price.value": { $lt: 600 },
  name: /TV/i
```

```
});

// Project name and price only
db.products.find(
  { category: "Electronics" },
  { name: 1, "price.value": 1, _id: 0 }
);
```

D. Update Product Stock

```
// Reduce stock by 1 when purchased
db.products.updateOne(
  { _id: ObjectId("665f4d7e8b3e6c1e24a7b3e2") },
  { $inc: { stock: -1 } }
);

// Add a "sale" tag
db.products.updateOne(
  { name: "4K Smart TV" },
  { $push: { tags: "sale" } }
);
```

E. Delete a Discontinued Product

```
db.products.deleteOne({
  name: "Legacy DVD Player"
});
```

6. Interactive Challenge / Mini-Project

Your Turn!

You're managing FastBite's menu database. Complete these tasks using MongoDB CRUD operations:

1. **Add a new vegan dish** called "Tofu Buddha Bowl" (cuisine: "Asian", price: $9.50, tags: ["vegan", "gluten-free"], available: true).

2. **Find all available vegan dishes** under $12, showing only their name and price.

3. **Update the price** of "Tofu Buddha Bowl" to $10.00 and add a "popular" tag.

4. **Delete** the dish "Old Special Soup" from the menu.

7. Common Pitfalls & Best Practices

Pitfall	Best Practice
Using update without operators	Always use `$set`, `$inc`, etc., not raw objects
Broad delete filters	Use unique identifiers or strict filters
No projection in find	Only return needed fields
Ignoring atomicity	Remember: each operation is atomic per document
No error handling	Always check results and handle errors

8. Optional: Programmer’s Workflow Checklist

- Validate input before insert/update.
- Use projections in queries to limit returned fields.
- Test update/delete filters with `find` before running.
- Use unique identifiers for critical updates/deletes.
- Handle errors and check operation results.