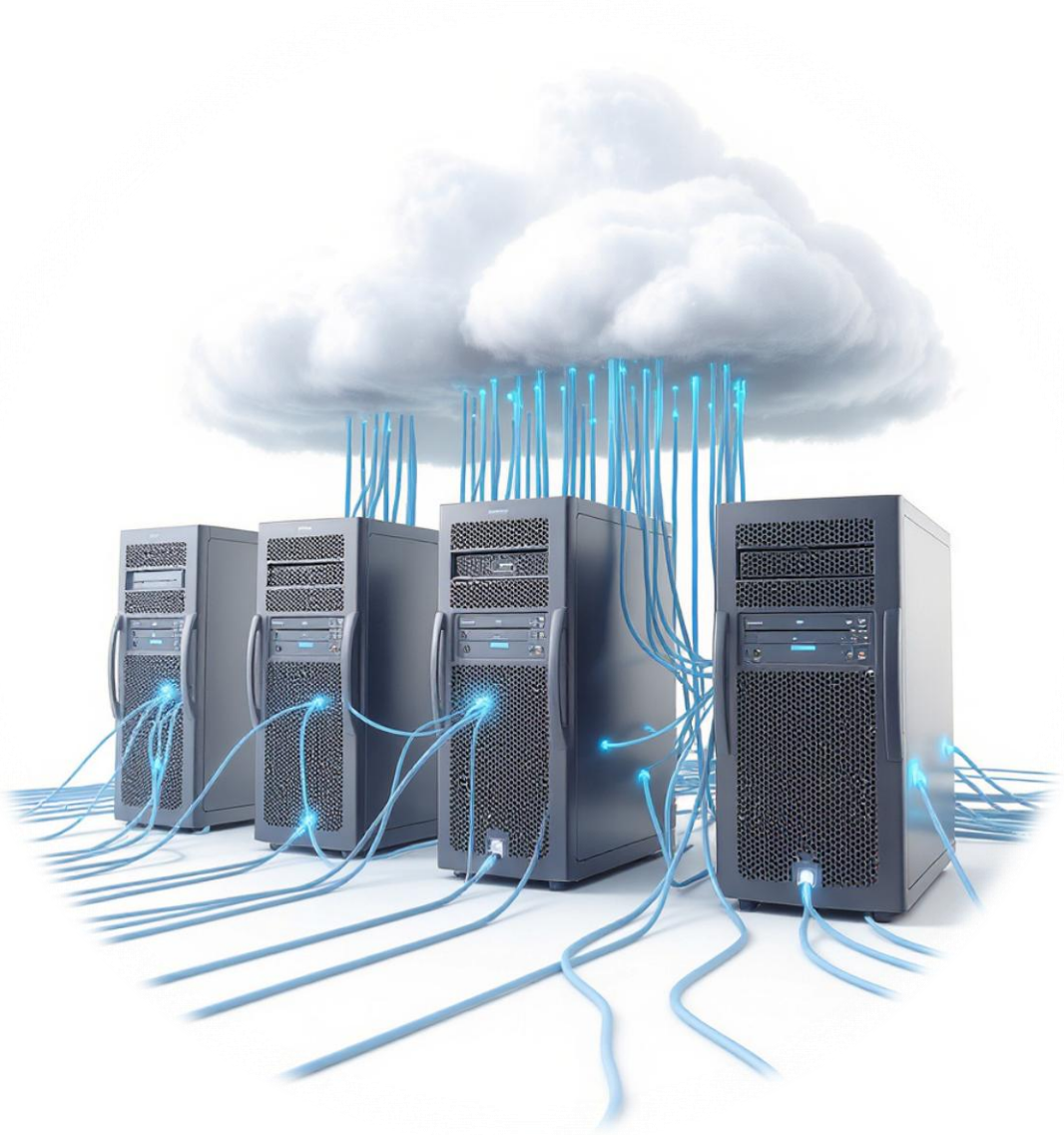
Chapter 12

The MongoDB DBMS

What is in This Chapter ?

This chapter begins with a very brief motivation for using databases and **Database Management Systems** (**DBMS**). We then discuss the **MongoDB** shell and how we can use it to create our own database to organize our data into collections. We examine some of the more common operations that can be performed from the terminal window using **mongosh**. We then consider how to use **MongoDB** within **Node.js**. We create a basic test program to connect to a database, perform some queries and then disconnect. We conclude with an example of a basic website that can perform product management by connecting an **Express** server running **PUG** code to our **MongoDB** database.



12.1 Database Introduction

Consider building a website that lets people create accounts, post comments, or save their favorite products. At first, it might seem simple enough to store that information in local files on our server. We can, for example, just write to a text file or a **JSON** file whenever new data comes in. This may work well if there are just a few intermittent accounts being created and used. However, as soon as multiple users start interacting with the site at the same time, things can become unpleasant …

- ✗ Handling asynchronous **file reads** and **writes** can become **complicated** and **error-prone**.

  - We have to keep track of callbacks or async/await logic for every read/write.
  - If multiple requests try to write to the file at once, race conditions can occur (i.e., one write overwriting another).
  - If an error happens mid-operation (e.g., file missing, permission denied, disk full), we need to catch and handle it properly or our whole app may crash.
  - As the number of operations grows, the code can become messy and harder to debug.
- ✗ Using synchronous file operations is considered **bad practice** because it **blocks** the entire server, slowing everything down.
- ✗ If we want to **search** or filter data (e.g., "find all users over 18"), we would need to write our own **custom code to parse** and scan the file, which is **inefficient** and **fragile**.

This is where databases (e.g., **MongoDB** , **Firebase Firestone**, **MySQL**, **SQLite**, **Microsoft SQL Server**, **Amazon DynamoDB**, etc..) can be useful.

A database is an organized collection of information that can be stored, updated, and searched efficiently.

Databases are designed to safely handle lots of users and data at once, making it fast to insert, update, and query information without conflicts/problems. Unlike local files, databases provide structure, scalability, and reliability … so our website can grow from a simple project to a full-fledged application that serves real users smoothly. But to make a database useful, we need …

> ***A Database Management System** (**DBMS**) **is special software that sits between our application and the database.***

Instead of us having to write raw file-handling code, the **DBMS** provides tools and commands to:

- ✓ Store data **efficiently** (i.e., in tables, documents, or other structures).
- ✓ **Manipulate** data (i.e., insert new records, update existing ones, or delete them safely).
- ✓ **Query** data (i.e., search and filter information quickly).
- ✓ **Control** access (i.e., handle multiple users at once without conflicts).
- ✓ Maintain **reliability** (i.e., recover from crashes, ensure data isn't corrupted, keep backups).

In short, the **DBMS** is what makes databases practical because it does the hard part (i.e., organizing and protecting data) while providing a simple, clean way for us developers to interact with it.

Beyond just storing data safely, databases also give us the following essential advantages:

- ✓ **Fast Queries** - we can search and filter data rapidly without writing custom code.
- ✓ **Reports & Insights** - data can be combined and turned into meaningful information (e.g., sales by region, most active users, etc..).
- ✓ **Automatic Optimizations** - handles indexing and performance tweaks "under the hood".
- ✓ **Reliability** (**ACID: A**tomicity, **C**onsistency, **I**solation, **D**urability) – guarantees that data remains valid even if something goes wrong (e.g., errors, crashes, or power loss).
- ✓ **Fault Tolerance** - can recover from failures or replicate data to keep things safe.

- ✓ **Concurrency** - multiple users can interact at the same time without corrupting information.
- ✓ **Scalability** - built to handle very large amounts of data as our app/site grows.

There are two main classes of **DBMS**:

- **Relational**: Stores data in structured tables with fixed rows and columns. Great for data with clear relationships (e.g., users ↔ orders ↔ products). Querying is done with ***SQL*** (***Structured Query Language***). Examples: **MySQL**, **PostgreSQL**, **Oracle**.
- **NoSQL** (i.e., **N**ot **O**nly **SQL**): Stores data in non-tabular formats and offer dynamic structure, making them easier to adapt as requirements change and they scale well to handle huge amounts of data and high traffic.

There are 4 types of **NoSQL** databases:

- **Document Databases**: stores data in documents that look like **JSON** objects.
- **Key-Value Databases**: each item is a key paired with a value.
- **Wide-Column Stores**: stores data in tables with rows and dynamic columns (sometimes called "two-dimensional key-value databases"). Different rows don't need to have the same columns.
- **Graph Databases**: stores data as nodes (things like people, places, or products) and edges (the relationships between them).

In this course, we will be using **MongoDB**, which is classified as a **NoSQL** (document-based) program that uses **JSON**-like documents with flexible structure, so each record can contain different types of information. It organizes data into collections of documents, with each document representing a single entity or object in the system. It is consistently ranked as the world's most popular **NoSQL** database.

12.2 Working with Databases in a MongoDB Shell

Let's get started using **MongoDB**. We need to first follow these steps:

1. Install **MongoDB Community Edition** by first downloading the package under **MongoDB Community Server** for your OS from here:

   https://www.mongodb.com/products/self-managed/community-edition

   Then it using all the default settings. Let's hope the latest version adheres to these notes.

2. Download **MongoDB Shell** (i.e., **mongosh**) from here (select **msi** package for windows):

   https://www.mongodb.com/docs/mongodb-shell/

   Install it using the default settings.

3. To run the **MongoDB Shell**, we need to open a **Command Prompt** terminal (not Powershell) and type **mongosh.** You will see something like what is shown below, although it may differ over time with each version update:

```
C:\Users\lanth\Documents\For School\Courses (Current)\COMP 2406\Chapter 12 - MongoDB\code\Lecture>mongosh
Current Mongosh Log ID: 68eea5f192818aabf2cebea3
Connecting to:          mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mon
gosh+2.5.8
Using MongoDB:          8.2.1
Using Mongosh:          2.5.8

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/

------
   The server generated these startup warnings when booting
   2025-10-14T15:27:32.192-04:00: Access control is not enabled for the database. Read and write access to data an
d configuration is unrestricted
------

test>
```

MongoDB organizes data in a hierarchy that is flexible and easy to understand:

MongoDB Server
└ Database
└ Collection
└ Document
└ Field

1. **Server / Instance**

   The **MongoDB** server (or instance) is the running process that holds our databases. We can connect to it via **mongosh** or through our application.

2. **Databases**

   Each server can have multiple databases (e.g., school, store, blog). Databases act like containers for collections.

3. **Collections**

   Each database contains one or more collections. A **collection** is roughly like a table in **SQL**, but it has **no fixed schema** (e.g., users, products, orders).

4. **Documents**

   A collection stores multiple **documents**. A **document** is a single record, stored as a **JSON**-like object. (e.g., **{** **"id"**: 1, **"name"**: **"Bob"**, **"email"**: **"bob@gmail.com"**, **"age"**: 22 **}**)

5. **Fields**

   Each document contains **fields** (i.e., key-value pairs) that hold data. (e.g., **"age"**: 22)

Creating and Modifying a MongoDB Database

There are different commands that we can use in **mongosh** to create and modify databases. Here are just a few of the more useful ones:

- **show dbs** – shows the current databases (three defaults exist: **admin**, **config** and **local**)

```
test> show dbs
admin   40.00 KiB
config  12.00 KiB
local   40.00 KiB
test>
```

- **db** – shows the current database name. (Note: upon startup, the current database is **`test`** which is like a "starting playground" where we can experiment until we create and switch to our own databases).

```
test> db
test
test>
```

- **use <dbName>** – switches to the database with name **<dbName>** if it already exists. Otherwise, it creates a database with the name **<dbName>** if it doesn't already exist (although the database is only actually created when we insert our first document, so it will not appear yet in the list of databases).

```
test> use store
switched to db store
store>
```

- **db.dropDatabase()** – removes all collections and documents inside the current database. Once executed, the action is irreversible, so use it carefully and make sure to use **db** to check that we are in the current database is the one that we want to delete. Note that the database still shows in the prompt, but it no longer exists on disk until we insert new data again.

```
store> db.dropDatabase()
{ ok: 1, dropped: 'store' }
store>
```

- **db.createCollection(<collectionName>)** – creates a collection with the given name in the current database. Alternatively, a collection will be created automatically as we add our first document.

```
store> db.createCollection("products")
{ ok: 1 }
store> db.createCollection("users")
{ ok: 1 }
store>
```

- **show collections** – shows all collections in the current database:

```
store> show collections
products
users
store>
```

- **db.<collectionName>. insertOne(<JSON Object>)** – adds a document to the specified collection named **<collectionName>** within the current database. Note that EVERY document MUST have a key called **_id**. If we accidentally provide a key name **id**, it will be added but then an additional **_id** key will also be automatically created. Note as well that the key names do not need quotes around them, but value strings do.

```
store> db.products.insertOne({_id:1, name:"Laptop", price:1200, stock:136 })
{
  acknowledged: true,
  insertedId: 1
}
store> db.products.insertOne({_id:2, name:"Mouse", price:25, stock:395 })
{
  acknowledged: true,
  insertedId: 2
}
```

```
store> db.products.insertOne({_id:2, name:"Pizza", price:9.99, stock:2 })
MongoServerError: E11000 duplicate key error collection: store.products index:
_id_ dup key: { _id: 2 }
```

- **db.<collectionName>.find()** – shows all documents in the collection named **<collectionName>** within the current database.

```
store> db.products.find()
[
  { _id: 1, name: 'Laptop', price: 1200, stock: 136 },
  { _id: 2, name: 'Mouse', price: 25, stock: 395 }
]
store>
```

- **db.<collectionName>.deleteOne({ key : value })** – deletes the first document that matches the given **key : value** filter.

```
store> db.products.deleteOne( {_id:2} )
{ acknowledged: true, deletedCount: 1 }
store> db.products.deleteOne( {_id:6} )
{ acknowledged: true, deletedCount: 0 }
store>
```

- **db.<collectionName>. insertMany([<JSON Obj1> , <JSON Obj2> , <JSON Obj3> , …])** – adds multiple documents to the specified collection named **<collectionName>** within the current database.

```
store> db.products.insertMany( [ {_id:3, name:"Pencils", price:1.79, stock:4352},
{_id:4, name:"Ruler", price:2.79, stock:2879}, {_id:9, name:"Mug", price:4.49,
stock:528} ] )
{ acknowledged: true, insertedIds: { '0': 3, '1': 4, '2': 9 } }
store>
```

- **db.<collectionName>.countDocuments()** – shows the number of documents in the collection named **<collectionName>** within the current database.

```
store> db.products.find()
[
  { _id: 1, name: 'Laptop', price: 1200, stock: 136 },
  { _id: 3, name: 'Pencils', price: 1.79, stock: 4352 },
  { _id: 4, name: 'Ruler', price: 2.79, stock: 2879 },
  { _id: 9, name: 'Mug', price: 4.49, stock: 528 }
]
store> db.products.countDocuments()
4
store>
```

Querying a MongoDB **Database**

Databases often contain a lot of data, but we usually only need a small portion at a time. To get exactly what we want, we use a …

query* = *a request to a database to retrieve specific data that matches certain criteria.

For example, we might want to see just a few product names and prices, sort them by price, or only look at products that are in stock. In **MongoDB**, we use the **find()** function to perform these queries and control which data is returned. Here are some examples of how this can be done effectively

- **db.<collectionName>.find({ key : { $operator : value } })** – shows all documents in the collection named **<collectionName>** within the current database whose **key** attribute has a **value** that matches the given query condition specified by **$operator**. When **.count()** is used at the end, it shows only the number of documents that matched.

```
store> db.products.find()
[
  { _id: 1, name: 'Laptop', price: 1200, stock: 136 },
  { _id: 3, name: 'Pencils', price: 1.79, stock: 4352 },
  { _id: 4, name: 'Ruler', price: 2.79, stock: 2879 },
  { _id: 5, name: 'Pizza', price: 9.99, stock: 2 },
  { _id: 6, name: 'Coke', price: 0.99, stock: 15 },
  { _id: 7, name: 'Chair', price: 345.99, stock: 83 },
  { _id: 8, name: 'Table', price: 429.97, stock: 17 },
  { _id: 9, name: 'Mug', price: 4.49, stock: 528 }
]
store> db.products.find({price:{$gt:10}})
[
  { _id: 1, name: 'Laptop', price: 1200, stock: 136 },
  { _id: 7, name: 'Chair', price: 345.99, stock: 83 },
  { _id: 8, name: 'Table', price: 429.97, stock: 17 }
]
store> db.products.find({price:{$lt:10}})
[
  { _id: 3, name: 'Pencils', price: 1.79, stock: 4352 },
  { _id: 4, name: 'Ruler', price: 2.79, stock: 2879 },
  { _id: 5, name: 'Pizza', price: 9.99, stock: 2 },
  { _id: 6, name: 'Coke', price: 0.99, stock: 15 },
  { _id: 9, name: 'Mug', price: 4.49, stock: 528 }
]
store> db.products.find({price:{$lt:10}}).count()
5
store>
```

Here are a few of the various **comparison** query operators available:

Operator	Description
`$eq`	=
`$ne`	!=
`$gt`	>
`$gte`	>=
`$lt`	<
`$lte`	<=
`$in [ ]`	matches values in the specified array of values
`$nin [ ]`	matches values not in the specified array of values

Queries can be combined using a comma:

```
store> db.products.find({price:{$lt:10}, stock:{$gt:500}})
[
  { _id: 3, name: 'Pencils', price: 1.79, stock: 4352 },
  { _id: 4, name: 'Ruler', price: 2.79, stock: 2879 },
  { _id: 9, name: 'Mug', price: 4.49, stock: 528 }
]
```

Queries can be combined for the same key field:

```
store> db.products.find({price:{$gt:10,$lt:500}})
[
  { _id: 7, name: 'Chair', price: 345.99, stock: 83 },
  { _id: 8, name: 'Table', price: 429.97, stock: 17 }
]
```

Values can be compared to items in an array:

```
store> db.products.find({name:{$in:["Chair", "Table"]}})
[
  { _id: 7, name: 'Chair', price: 345.99, stock: 83 },
  { _id: 8, name: 'Table', price: 429.97, stock: 17 }
]
```

We can use regular expressions to see if there is a partial match for a string:

```
store> db.products.find()
[
  { _id: 1, name: 'Laptop', price: 1200, stock: 136 },
  { _id: 3, name: 'Pencils', price: 1.79, stock: 4352 },
  { _id: 4, name: 'Ruler', price: 2.79, stock: 2879 },
  { _id: 5, name: 'Pizza', price: 9.99, stock: 2 },
  { _id: 6, name: 'Coke', price: 0.99, stock: 15 },
  { _id: 7, name: 'Chair', price: 345.99, stock: 83 },
  { _id: 8, name: 'Table', price: 429.97, stock: 17 },
  { _id: 9, name: 'Mug', price: 4.49, stock: 528 },
  { _id: 10, name: 'Red Ball', price: 2.99, stock: 24 },
  { _id: 11, name: 'Blue Ball', price: 2.99, stock: 46 },
  { _id: 12, name: 'Green Ball', price: 2.99, stock: 34 }
]
```

```
store> db.products.find({name:{$regex:"Ball"}})
[
  { _id: 10, name: 'Red Ball', price: 2.99, stock: 24 },
  { _id: 11, name: 'Blue Ball', price: 2.99, stock: 46 },
  { _id: 12, name: 'Green Ball', price: 2.99, stock: 34 }
]
store>
```

Here are a few of the various **logical** query operators available:

Operator	Description
`$and`	matches the conditions of all clauses
`$not`	do not match the conditions of the query
`$nor`	fail to match the conditions of all clauses
`$or`	match the conditions of either clause

Here is an example of finding cheap items that have a lot of stock:

```
store> db.products.find ({$and:[{price:{$lt:5}}, {stock:{$gt:100}}]})
[
  { _id: 3, name: 'Pencils', price: 1.79, stock: 4352 },
  { _id: 4, name: 'Ruler', price: 2.79, stock: 2879 },
  { _id: 9, name: 'Mug', price: 4.49, stock: 528 }
]
```

Here is an example of finding cheap items OR ones that are low in stock:

```
store> db.products.find ({$or:[{price:{$lt:2.99}}, {stock:{$lt:20}}]})
[
  { _id: 3, name: 'Pencils', price: 1.79, stock: 4352 },
  { _id: 4, name: 'Ruler', price: 2.79, stock: 2879 },
  { _id: 5, name: 'Pizza', price: 9.99, stock: 2 },
  { _id: 6, name: 'Coke', price: 0.99, stock: 15 },
  { _id: 8, name: 'Table', price: 429.97, stock: 17 }
]
store>
```

Here is an example of finding items that are neither cheap **NOR** low in stock:

```
store> db.products.find ({$nor:[{price:{$lt:2.99}}, {stock:{$lt:20}}]})
[
  { _id: 1, name: 'Laptop', price: 1200, stock: 136 },
  { _id: 7, name: 'Chair', price: 345.99, stock: 83 },
  { _id: 9, name: 'Mug', price: 4.49, stock: 528 },
  { _id: 10, name: 'Red Ball', price: 2.99, stock: 24 },
  { _id: 11, name: 'Blue Ball', price: 2.99, stock: 46 },
  { _id: 12, name: 'Green Ball', price: 2.99, stock: 34 }
]
store>
```

We can also handle nested/embedded objects by using the dot notation. Quotes are required around the attribute parameters. Here is an example of how to do this:

```
store> db.products.insertMany([
  {
    _id: 22, name: "Tablet A", price: 499.99, stock: 25,
    details: {
      manufacturer: { name: "TechBrand", country: "Korea", warrantyYears: 1 },
      specs: { screen: "10 inch", storage: "128GB", battery: "7000mAh" }
    }
  },
  {
    _id: 23, name: "Tablet B", price: 599.99, stock: 30,
    details: {
      manufacturer: { name: "GigaTech", country: "USA", warrantyYears: 2 },
      specs: { screen: "11 inch", storage: "256GB", battery: "8000mAh" }
    }
  },
  {
    _id: 24, name: "Tablet C", price: 699.99, stock: 18,
    details: {
      manufacturer: { name: "SmartWorld", country: "Japan", warrantyYears: 1 },
      specs: { screen: "12.5 inch", storage: "256GB", battery: "9000mAh" }
    }
  }
])
{ acknowledged: true, insertedIds: { '0': 22, '1': 23, '2': 24 } }
store> db.products.find({"details.specs.storage": "128GB"})
[
  {
    _id: 22,
    name: 'Tablet A',
    price: 499.99,
    stock: 25,
    details: {
      manufacturer: { name: 'TechBrand', country: 'Korea', warrantyYears: 1 },
      specs: { screen: '10 inch', storage: '128GB', battery: '7000mAh' }
    }
  }
]
store>
```

When working with a database, we often don't need ***all*** the information stored in a document. For example, suppose we are shopping online and just want to see the **name** and **price** of items (i.e., we may not care about their **manufacturer**, **warrantyYears**, **_id**, etc..). If the database sent us all the details every time, it would be wasteful (i.e., more data to transfer, more memory to hold it, and more noise for us to filter through).

Thankfully, **MongoDB** allows us to use …

> ***A projection is a way to select which fields of a document are returned by a query, including some and excluding others.***

A projection let's us say: *"I only want these fields, and nothing else."* It helps keep queries fast, makes our results easier to read, and reduces the amount of data traveling across the network. Projections are especially important in large systems, where millions of users are querying massive collections.

A projection is supplied as the second parameter of the **find()** command:

- **db.<collectionName>.find({ key : value } , { key1:flag1, key2:flag2, key3:flag3 … })** – shows all documents in the collection named **<collectionName>** within the current database whose **key** attribute has the specified **value**. The **flags** above are either **0** or **1**. The only **key: value** pairs that are returned are those in which their **flag** is set to **1**. The **_id** field is returned by default, but can be disabled by setting its flag to **0** (i.e., **_id:0**). Alternatively, we can specify a **0** for all fields we don't want, and all others will be included.

```
store> db.products.find({}, {name: 1, price:1, _id:0}))
[
  { name: 'Laptop', price: 1200 },
  { name: 'Pizza', price: 9.99 },
  { name: 'Coke', price: 0.99 },
  { name: 'Chair', price: 345.99 },
  { name: 'Table', price: 429.97 },
  { name: 'Red Ball', price: 2.99 },
  { name: 'Blue Ball', price: 2.99 },
  { name: 'Green Ball', price: 2.99 },
  { name: 'Pencils', price: 1.79 },
  { name: 'Ruler', price: 2.79 },
  { name: 'Mug', price: 4.49 },
  { name: 'Tablet A', price: 499.99 },
  { name: 'Tablet B', price: 599.99 },
  { name: 'Tablet C', price: 699.99 }
]
store> db.products.find({}, {name:0, price:0, _id:0, details:0})
[
  { stock: 136 },  { stock: 2 },
  { stock: 15 },   { stock: 83 },
  { stock: 17 },   { stock: 24 },
  { stock: 46 },   { stock: 34 },
  { stock: 4352 }, { stock: 2879 },
  { stock: 528 },  { stock: 25 },
  { stock: 30 },   { stock: 18 }]
store>
```

There are some other functions that we can apply after the **find()** command that are very useful for pagination:

- **db.<collectionName>.find(… }.limit(<n>)** – returns the first **<n>** results that match the specified query documents.

- **db.<collectionName>.find(… }.skip(<m>)** – skips over the first **<m>** results that match the specified query documents and returns the rest.

- **db.<collectionName>.find(… }.skip(<m>).limit(<n>)** – skips over the first **<m>** results that match the specified query documents and then returns the next **<n>** results.

```
store> db.products.find().limit(5)
[
  { _id: 1, name: 'Laptop', price: 1200, stock: 136 },
  { _id: 3, name: 'Pencils', price: 1.79, stock: 4352 },
  { _id: 4, name: 'Ruler', price: 2.79, stock: 2879 },
  { _id: 5, name: 'Pizza', price: 9.99, stock: 2 },
  { _id: 6, name: 'Coke', price: 0.99, stock: 15 }
]
store> db.products.find().skip(2).limit(5)
[
  { _id: 4, name: 'Ruler', price: 2.79, stock: 2879 },
  { _id: 5, name: 'Pizza', price: 9.99, stock: 2 },
  { _id: 6, name: 'Coke', price: 0.99, stock: 15 },
  { _id: 7, name: 'Chair', price: 345.99, stock: 83 },
  { _id: 8, name: 'Table', price: 429.97, stock: 17 }
]
store>
```

Whenever we want to quickly grab just one document, rather than dealing with a full list of results, we can use **findOne(…)** instead of **find(…)**. This is useful if we are looking up a specific user by their unique username or email address. **findOne()** is better because it stops after it finds the match, while **find()** keeps looking for all matches … and in this particular situation, we know that there is only one match, so why waste time looking for more?

However, we can always use **find().limit(1)** to get just one product as well. The difference is in the format returned. **findOne()** returns one document, whereas **find().limit(1)** returns a cursor (which is essentially *a pointer to* the results … which must be handled differently).

- **db.<collectionName>.find(… }.sort({ key : flag})**
  – sorts the results in order of the specified **key**. If the **flag** is **1**, it sorts in ascending order. If the **flag** is **-1**, it sorts in descending order.

```
store> db.products.find({price:{$lt:450}}).sort({stock: 1})
[
  { _id: 5, name: 'Pizza', price: 9.99, stock: 2 },
  { _id: 6, name: 'Coke', price: 0.99, stock: 15 },
  { _id: 8, name: 'Table', price: 429.97, stock: 17 },
  { _id: 10, name: 'Red Ball', price: 2.99, stock: 24 },
  { _id: 12, name: 'Green Ball', price: 2.99, stock: 34 },
  { _id: 11, name: 'Blue Ball', price: 2.99, stock: 46 },
  { _id: 7, name: 'Chair', price: 345.99, stock: 83 },
  { _id: 9, name: 'Mug', price: 4.49, stock: 528 },
  { _id: 4, name: 'Ruler', price: 2.79, stock: 2879 },
  { _id: 3, name: 'Pencils', price: 1.79, stock: 4352 }
]
store>
```

Here is a link to some documentation that describes the various query operators:

https://www.mongodb.com/docs/manual/reference/mql/expressions/

Updating & Deleting in a MongoDB Database

In addition to querying the database, we sometimes need to **replace** an old document with a new one (e.g., when a discontinued product is replaced by a newer version). At other times, we may want to **update** only certain fields of a document, such as changing the price or stock quantity of a product. **MongoDB** provides built-in functions that make both replacing and updating documents straightforward and efficient.

All modifications in **MongoDB** are ***atomic***, meaning each change is completed fully before another begins. This ensures that two processes cannot modify the same document at the same time, preventing conflicts and avoiding data corruption that could occur if multiple processes tried to write to the same file simultaneously.

Here is how to do an update of a document:

- **db.<collectionName>.updateOne(<filter>, <update>, <options>)** – updates a single document in a collection that matches the given filter. The **<filter>** identifies the specific document. **<update>** defines how to modify the document and **<options>** are optional settings.

```
store> db.products.findOne({_id:1})
{ _id: 1, name: 'Laptop', price: 1200, stock: 136 }
store> db.products.updateOne({_id:1}, {$set: {price: 1079}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
store> db.products.findOne({_id:1})
{ _id: 1, name: 'Laptop', price: 1079, stock: 136 }
store> db.products.updateOne({_id:1}, {$inc: {stock: -1}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
store> db.products.findOne({_id:1})
{ _id: 1, name: 'Laptop', price: 1079, stock: 135 }
```

```
store> db.products.updateOne({_id:1}, {$inc: {stock: 100}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
store> db.products.findOne({_id:1})
{ _id: 1, name: 'Laptop', price: 1079, stock: 235 }
store>
```

Notice the resulting object that contains information about how many documents were modified, among other information. We can use this information to decide what response to send. Notice the **upsertedCount** field in the result. It shows how many documents were inserted because of an upsert (i.e., "update insert" = upsert) operation. If we include **{ upsert: true }**, in the **<options>** parameter, **MongoDB** will create and insert a new document when no existing document matches the specified **<filter>**.

We can include multiple update operators at the same time, separated by commas

(e.g., **{$set:{price: 500},$inc:{stock: 100}}**).

We can include multiple fields for each update operator, separated by commas

(e.g., **{$set:{price: 300, stock: 50}}**).

We can handle nested values using the dot operator and quotations:

(e.g., **{$set:{"details.manufacturer.warantyYears":5}}**).

These are some additional useful update operators:

- **$rename** – renames a field in a document
- **$unset** – removes the specified field from a document

Look here for a more thorough list of update operators:

https://www.mongodb.com/docs/manual/reference/mql/update/

There is also an **updateMany()** function that works similar to **insertMany()**:

- **db.<collectionName>.updateMany(<filter>, <update>, <options>)** – updates all documents in a collection that match the given filter.

```
store> db.products.find({stock:{$gt:50}})
[
  { _id: 1, name: 'Laptop', price: 1079, stock: 235 },
  { _id: 7, name: 'Chair', price: 345.99, stock: 83 },
  { _id: 3, name: 'Pencils', price: 1.79, stock: 4352 },
  { _id: 4, name: 'Ruler', price: 2.79, stock: 2879 },
  { _id: 9, name: 'Mug', price: 4.49, stock: 528 }
]

store> db.products.updateMany({$and:[{stock:{$gt:50}}, {price:{$gt:100}}]},
{$inc:{price:-100}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 2,
  modifiedCount: 2,
  upsertedCount: 0
}

store> db.products.find({stock:{$gt:50}})
[
  { _id: 1, name: 'Laptop', price: 979, stock: 235 },
  { _id: 7, name: 'Chair', price: 245.99, stock: 83 },
  { _id: 3, name: 'Pencils', price: 1.79, stock: 4352 },
  { _id: 4, name: 'Ruler', price: 2.79, stock: 2879 },
  { _id: 9, name: 'Mug', price: 4.49, stock: 528 }
]
store>
```

- **db.<collectionName>.replaceOne(<filter>, <replacementObject>)** – replace a single document in a collection that matches the given **<filter>** with the **<replacementObject>**. This is useful, for example, if we need to replace a discontinued product with a newer version of the product where many of the attributes are different.

```
store> db.products.find({_id:24})
[
  {
    _id: 24,
    name: 'Tablet C',
    price: 699.99,
    stock: 18,
    details: {
      manufacturer: { name: 'SmartWorld', country: 'Japan', warrantyYears: 1 },
      specs: { screen: '12.5 inch', storage: '256GB', battery: '9000mAh' }
    }
  }
]
store> db.products.replaceOne({ _id: 24 }, { _id: 24, name: 'Tablet C+', price:
799.99, stock: 50,
...     details: {
...       manufacturer: { name: 'SmartWorld', country: 'Japan', warrantyYears: 2
},
```

```
...       specs: { screen: '12.5 inch', storage: '512GB', battery: '9500mAh' }
...     }
...   })
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
store> db.products.find({_id:24})
[
  {
    _id: 24,
    name: 'Tablet C+',
    price: 799.99,
    stock: 50,
    details: {
      manufacturer: { name: 'SmartWorld', country: 'Japan', warrantyYears: 2 },
      specs: { screen: '12.5 inch', storage: '512GB', battery: '9500mAh' }
    }
  }
]
store>
```

The **...** in the above example is from the shell's prompt. It is a visual queue to show us that we are typing in a multi-line command.

- **db.<collectionName>.deleteOne(<filter>)** – delete a single document in a collection that matches the given **<filter>**.

- **db.<collectionName>.deleteMany(<filter>)** – delete all documents in a collection that match the given **<filter>**.

```
store> db.products.deleteOne({price:{$gt: 400}})
{ acknowledged: true, deletedCount: 1 }
store> db.products.deleteOne({_id:5})
{ acknowledged: true, deletedCount: 1 }
store> db.products.deleteMany({_id:{$gt:10}})
{ acknowledged: true, deletedCount: 4 }
store> db.products.find()
[
  { _id: 3, name: 'Pencils', price: 1.79, stock: 4352 },
  { _id: 4, name: 'Ruler', price: 2.79, stock: 2879 },
  { _id: 6, name: 'Coke', price: 0.99, stock: 15 },
  { _id: 7, name: 'Chair', price: 345.99, stock: 83 },
  { _id: 8, name: 'Table', price: 429.97, stock: 17 },
  { _id: 9, name: 'Mug', price: 4.49, stock: 528 },
  { _id: 10, name: 'Red Ball', price: 2.99, stock: 24 }
]
store>
```

At this point, we have just looked at the basics of **MongoDB** from the shell window. There are a lot of operations that we can perform on the database, but we shouldn't worry about trying to know and understand them all. We should, however, get comfortable with using the documentation to build queries to achieve specific goals.

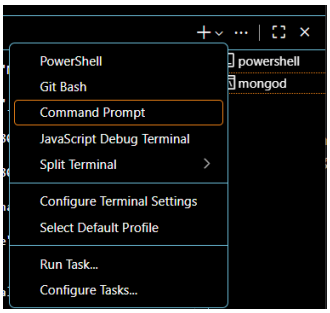
Now let's see how to incorporate **MongoDB** into a **Node.js** web application.

12.3 MongoDB in Node.js

Before we can use **MongoDB** with **Node.js**, we need to get the **Mongo daemon** running. I am not sure how to describe this for iOS or Linux … but I will describe the process for Windows.

We have to first decide where we want **MongoDB** to store its files related to our databases. Typically, we just choose a folder and then tell the **Mongo daemon** where it is. Let's do this:

1. Make a subfolder called **db** within a **data** folder at the root of our system (i.e., **C:\data\db**). We will then tell the **Mongo daemon** to use that folder.
2. Identify the location that the **Mongo daemon** executable (i.e., **mongod.exe**) was installed into (e.g., **C:\Program Files\MongoDB\Server\8.2\bin\mongod.exe**).
3. Open up a **Command Prompt** window and navigate to the folder from step **2** that contains the **mongod.exe** file (Note: The **Powershell** terminal window in **VScode** does not have the right permissions). We can choose it from the arrow beside the **+** in **VSCode** →

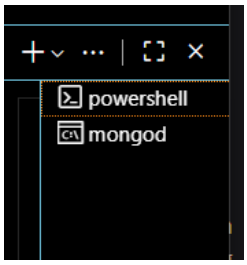


4. Run this to start the daemon: **mongod.exe --dbpath=C:\data\db**. We should see a bunch on "non-sensical" stuff appear that looks like this:

{"t":{"$date":"2025-08-20T16:29:21.438-04:00"},"s":"I", "c":"CONTROL", "id":23285,
"ctx":"thread1","msg":"Automatically disabling TLS 1.0, to force-enable TLS 1.0 specify --sslDisabledProtocols 'none'"}
{"t":{"$date":"2025-08-20T16:29:21.441-04:00"},"s":"I", "c":"CONTROL", "id":5945603, "ctx":"thread1","msg":"Multi threading initialized"}
{"t":{"$date":"2025-08-20T16:29:21.441-04:00"},"s":"I", "c":"NETWORK", "id":4648601, "ctx":"thread1","msg":"Implicit TCP FastOpen unavailable. If TCP FastOpen is required, set at least one of the related parameters","attr":{"relatedParameters":["tcpFastOpenServer","tcpFastOpenClient","tcpFastOpenQueueSize"]}}
{"t":{"$date":"2025-08-20T16:29:21.442-04:00"},"s":"I", "c":"NETWORK", "id":4915701, "ctx":"thread1","msg":"Initialized wire specification","attr":{"spec":{"incomingExternalClient":{"minWireVersion":0,"maxWireVersion":25},"incomingInternalClient":{"minWireVersion":0,"maxWireVersion":25},"outgoing":{"minWireVersion":6,"maxWireVersion":2
…

The code daemon will still be running … indication that the **MongoDB** server is ready and listening for us to talk to it (as clients).

5. We will need to use a different terminal window to run our server. We will open up another terminal now … it can be a powershell window this time. A powershell window may already be running in **VScode**, but if not, we can press the **+** off to the right side of the terminal to make a new one. We can swap between them by selecting them from the list … as we can see from the image here on the right →



6. From that new terminal, we can navigate to the directory that will contain our server code (e.g., **C:\Users\ianth\Documents\COMP 2406\code\MyWebsite**) and install the **mongodb** module in **NPM**: **npm install mongodb**

Now we are ready to use the database in our code. Keep in mind that steps **3 & 4** will be necessary every time that we want to run our server because the **Mongo server daemon** must also be running at all times for us to be able to access our database. Our server will become a client of the **MongoDB** server.

Let's first look at how to write a **JavaScript** program (i.e., **test_mongo_connect.js**) that communicates with **MongoDB** and then later we will see how to do this with a server.

Now, in our course examples, we usually have a simple, single server, so we only need one module that talks to the database. In that case, it would be fine to put both the server code and the database code together in the same file. However, in larger applications the server-to-database communication code is often required across multiple modules (e.g., **users.js**, **products.js**, **auth.js**, etc.). Each of these modules may need to access different collections in the same database.

If every module managed its own connection, we would end up with repeated code and possibly multiple unnecessary connections to the database. By moving the connection logic into a separate file (e.g., **db.js**), we centralize that responsibility so that the whole application shares a single connection. This also makes the system easier to maintain. If, for example, the connection details change (e.g., the database **URL** or error-handling strategy), we only need to make updates in one location and the entire app benefits.

So, what does our database module (**db.js**) look like? It needs to first get/require the **MongoClient** class from the **mongodb** module so that we can connect and talk to the database server. Then we can set up variables to indicate the address of the **MongoDB** server as well as the name of our database.

Finally, we will maintain a `client` object that will represent our connection to the database and a `db` variable that will represent the database that we are connected to … as follows:

```
// Get the MongoClient class from the mondodb module
const { MongoClient } = require("mongodb");

// Connection URL for the MongoDB server (default port for Mongo is 27017)
const url = "mongodb://localhost:27017";
const dbName = "store";        // This is the name of our database

// Create a MongoClient instance
const client = new MongoClient(url);

let db = null;   // The database connection (null upon start)
```

Now, we will need to use the **MongoClient** to *connect to* & *disconnect from* the **MongoDB** server. We can set things up so that whenever we need to access something from the database, we call a function to connect to it, and get back a database object that we can work with. Then we can call a disconnect function to release the connection.

The latest version of **MongoDB** requires us to use the `async`/`await` approach that we discussed a while ago in our **AJAX** chapter. Here is the code that we will add to our **db.js** file:

```
// Connect to the database
async function connectToDatabase() {
    if (!db) { // If not already connected, connect now
        await client.connect();
        console.log("Connected to MongoDB: " + dbName);
        db = client.db(dbName);
    }
    return db;
}

// Disconnect from the database
async function disconnectFromDatabase() {
    await client.close();
    db = null;   // Reset so next call to connectToDatabase() works properly
    console.log("Disconnected from MongoDB: " + dbName);
}

module.exports = { connectToDatabase, disconnectFromDatabase };
```

Notice how the connection code checks to see if the database is already set before connecting again. This prevents someone from trying to connect again after they are already connected. The `db` object is returned to the caller so that we can use it. Closing the connection is also quite simple.

Take notice that these two function make use of `await`. So, the function that we call these from will have to have `async` in front of it.

Finally, notice that we are exporting the two functions. So, to use them in our code, we will need to do something like this:

```
const { connectToDatabase, disconnectFromDatabase } = require("./db");
```

In typical real-world **Node.js** apps, we wouldn't call `disconnectFromDatabase()` after every query. That function is mainly used when the app/server is shutting down. Therefore, in the `require()` above, we will often see modules only importing the `connectToDatabase()` function.

Notice as well that the code above assumes that **db.js** is in the same folder as wherever we write this code.

But wait!

Our functions do not have any error-checking in them! What if something goes wrong (e.g., someone forgot to start the **MongoDB** server)?

Let's add some error-handling code.

Here is a better version of these functions …

```
// Connect to the database
async function connectToDatabase() {
    if (!db) { // If not already connected, connect now
        try {
            await client.connect();
            db = client.db(dbName);
            console.log("Connected to MongoDB: " + dbName);
        } catch (err) {
            console.error("Error connecting to MongoDB:", err);
            throw err; // re-throw this error so that the caller knows it failed
        }
    }
    return db;
}

// Disconnect from the database
async function disconnectFromDatabase() {
    try {
        await client.close();
        db = null;  // Reset so next call to connectToDatabase() works properly
        console.log("Disconnected from MongoDB: " + dbName);
    } catch (err) {
        console.error("Error disconnecting from MongoDB:", err);
    }
}
```

At this point, we have a nice clean module that allows us to connect to one database on one **MongoDB** server. If we wanted to use multiple databases at the same time, we would need to maintain multiple database variables or use an array to hold them. If we needed to communicate with multiple database servers, this would require us to have multiple clients, each with their own **connect**/**close** functionality. In all cases, we will want to reuse connections, not create a new one every time. In this course, we will keep things simple with one database on one server.

Here is a basic test program that will use our **db.js** module to connect to a database, display its collections and then disconnect from the database:

```
// Get the database connect/disconnect functions
const { connectToDatabase, disconnectFromDatabase } = require("./db");

// We made a main() function because we are required to use "async"
async function main() {
    try {
        const db = await connectToDatabase();

        const collections = await db.collections();
        console.log("Collections:", collections.map(c => c.collectionName));

    } catch (err) {
        console.error("Error ... something went wrong:", err);
    } finally {
        await disconnectFromDatabase();
    }
}
main();
```

Notice that the code is simple and clean with the use of **db.js**.

Once we connect to the database, we can begin to use it to do interesting things. We have already seen how to do various operations with the **MongoDB** shell **mongosh**. As it turns out, the same functionality is available for **MongoDB** under **Node.js**.

Here is a comparison of the **mongosh** functionality with **Node.js**. We will notice that most function calls require the use of **await** since they are asynchronous calls that return promises.

```
> mongosh
```

```
const client = new MongoClient("mongodb://localhost:27017");
await client.connect();
```

```
> use store
```

```
const db = client.db("store");
```

```
> show collections
```

```
const collections = await db.collections(); // returns an array of Collection objects
```

```
> db.users.findOne({name: "Steve"})
```

```
const doc = await db.collection("users").findOne({name: "Steve"});
```

```
> db.users.find({})
```

```
const docs = await db.collection("users").find().toArray(); // .toArray() executes the query
```

A little bit of explanation is needed for this one. The call to **.find()** does not immediately return the documents. Instead, it returns a **Cursor** object … which is like a *pointer* to the matching documents on the server. It allows us to iterate over the results without loading everything into memory at once. The call to **.toArray()** actually executes the query on the server, fetches all the documents, and then returns them as a **JavaScript** array. There are multiple ways to use the cursor object. Here are a few:

```
const cursor = db.collection("users").find();
const firstDoc = await cursor.next(); // gets the first object from the collection
```

```
const cursor = db.collection("users").find();
let doc;
while ((let doc = await cursor.next()) !== null) { // gets one object at a time
    console.log(doc);
}
```

```
const cursor = db.collection("users").find({});
for await (const doc of cursor) {  // iterates through each object, one at a time
    console.log(doc);
}
```

The last one (above) is the preferred pattern, especially when we are doing multiple operations with each document, including other async calls.

Here are some more comparisons of functions with **mongosh**. Most of these return a value with useful data about the operation that was completed.

```
> db.users.insertOne({name: "Abdul"})
```

```
const result = await db.collection("users").insertOne({name: "Abdul"});
```

In the above code, `result` will hold a **JavaScript** object like this:

```
{ acknowledged: true, insertedId: ObjectId("66cc2f42e317a841dce293b7") }
```

in which we can determine if the operation was a success and even get back the ID of the inserted item (e.g., by using `result.acknowledged` or `result.insertedId`).

```
> db.users.insertMany([{name:"Chen"}, {name:"Hans"}])
```

```
const result = await db.collection("users").insertMany([{name:"Chen"}, {name:"Hans"}]);
```

Similar to inserting one item, `result` will return a **JavaScript** object but it will also list multiple IDs:

```
{
  acknowledged: true,
  insertedCount: 2,
  insertedIds: {
      '0': ObjectId("…"),
      '2': ObjectId("…")
  }
}
```

The other functions also have return objects that let us query the result … but I won't list them here.

```
> db.users.updateOne({name:"Martha"}, {$set:{age:20}})
```

```
const result = await db.collection("users").updateOne({name:"Martha"}, {$set:{age:20}});
```

```
> db.users.updateMany({age:{$lt:18}}, {$set:{minor:true}})
```

```
const result = await db.collection("users").updateMany({age:{$lt:18}}, {$set:{minor:true}});
```

```
> db.users.deleteOne({name:"Priyal"})
```

```
const result = await db.collection("users").deleteOne({name:"Priyal"});
```

```
> db.users.deleteMany({minor:true})
```

```
const result = await db.collection("users").deleteMany({minor:true});
```

```
> db.users.countDocuments({})
```

```
const count = await db.collection("users").countDocuments({});
```

> db.users.drop()

```
const result = await db.collection("users").drop();
```

> db.dropDatabase()

```
const result = await db.dropDatabase();
```

Here is a simple test program that connects to the database, adds a few products to a **products** database and then performs some queries (assume though that we changed the name of the database in **db.js** as follows: `const db = client.db("myDatabase");` because it uses some of the same product ideas as the store we had before and we want it to start fresh):

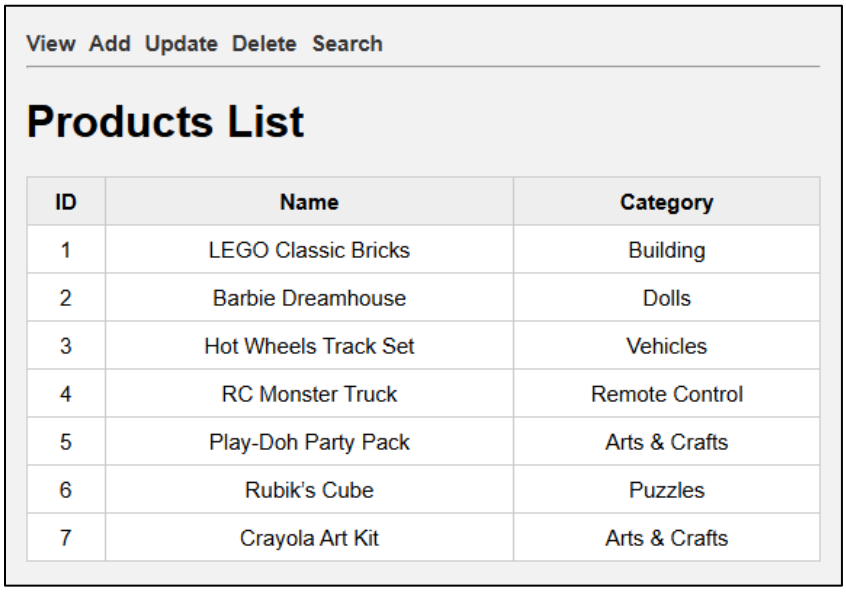
```
// This code will connect to a database add some toys to a products collection and then perform
// some queries/updates and then quit.

// Get the database connect/disconnect functions
const { connectToDatabase, disconnectFromDatabase } = require("./db");

// Here are some toys that we will insert
const toys = [
  { _id: 1, name: "LEGO Classic Bricks", category: "Building", price: 29.99, stock: 120, ageRange: "4+"},
  { _id: 2, name: "Barbie Dreamhouse", category: "Dolls", price: 199.99, stock: 25, ageRange: "3+"},
  { _id: 3, name: "Hot Wheels Track Set", category: "Vehicles", price: 49.99, stock: 75, ageRange: "5+"},
  { _id: 4, name: "NERF Blaster Elite", category: "Outdoor", price: 39.99, stock: 60, ageRange: "8+"},
  { _id: 5, name: "Play-Doh Party Pack", category: "Arts&Crafts", price: 12.99, stock: 200, ageRange: "3+"},
  { _id: 6, name: "Rubik's Cube", category: "Puzzles", price: 9.99, stock: 150, ageRange: "6+"}
];


// We made a main() function because we are required to use "async"
async function main() {
    try {

        const db = await connectToDatabase();

        // Create a products collection
        const products = db.collection("products");

        // Insert the toys in the above-defined array
        let result = await products.insertMany(toys);
        console.log(result.insertedCount + " products were added.");


        // Display all added products
        let prods = await products.find();

        console.log("Here are the products that were added:");
        for await (const p of prods) {
            console.log(p);
        }
```

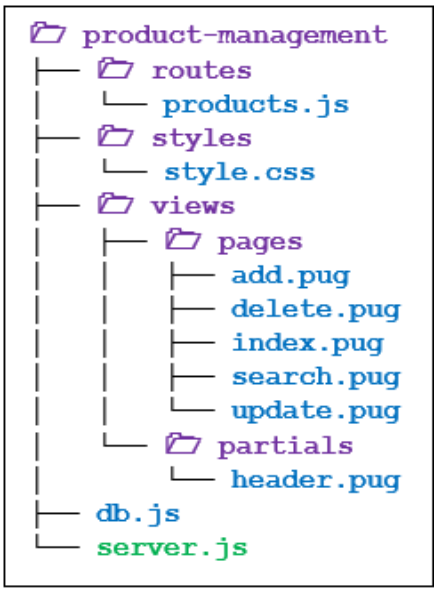
```
        // Display all products with a price under $30
        let cheap = await products.find({price:{$lt:30}});
        console.log("Here are the products under $30:");
        for await (const p of cheap) {
            console.log(p);
        }

        // Update the price of the Barbie Dreamhouse
        result = await products.updateOne({name:"Barbie Dreamhouse"}, {$set: {price: 179.99}});
        if (result.acknowledged)
            console.log("Price has been updated for Barbie Dreamhouse");
        else
            console.log("Price has NOT been updated for Barbie Dreamhouse");

        // Update the price of the Barbie Dreamhouse
        let product = await products.findOne({name:"Barbie Dreamhouse"}).toArray();
        console.log("Price is now at $" + product[0].price);

        // Alternatively, we can do the following to use the cursor instead of the lines above:
        //     let product = await products.find( { name: "Barbie Dreamhouse" });
        //     console.log("Price is now at $" + (await product.next()).price);


        // Replace the NERF Blaster Elite with a new prouct (using the same _id)
        result = await products.replaceOne({_id: 4}, {_id: 4, name: "RC Monster Truck",
                                    category: "Remote Control", price: 59.99, stock: 40, ageRange: "6+"});
        if (result.acknowledged)
            console.log("NERF Blaster Elite has been replaced by RC Monster Truck");
        else
            console.log("NERF Blaster Elite has NOT been replaced");

        // Display all  products
        prods = await products.find();
        console.log("Here are the products in the database:");
        for await (const p of prods) {
            console.log(p);
        }

    } catch (err) {
        console.error("Error ... something went wrong:", err);
    } finally {
        await disconnectFromDatabase();
    }
}

main();
```

The code shows that we can connect to the database and perform various operations on the database in a similar way to interacting with the database via the **mongosh**. The above code will run once, but if we try to run it again, it will throw an error because we would be trying to add 6 more products with the same **_id** values as the ones already in there. This was just a proof-of-concept program. We would not typically set up our code like this. Most interactions will be user-driven.

12.4 Products Management Site Example

Let's look at an example of making a simple website to manage the products in our database. The folder hierarchy will be as shown on the right below. We will use **Express** and **PUG** to make our dynamic pages. The main site will have the simple look as shown on the left below:





The navigation bar will allow us to **View**, **Add**, **Update**, **Delete** and **Search** for products. The products will be the ones that are currently in the **products** collection of the **myDatabase** database that we made earlier. We will use the same **db.js** module that we created earlier to connect to the database. We will set up our routes under a products router so that we use the following **URL**s to access the appropriate behavior:

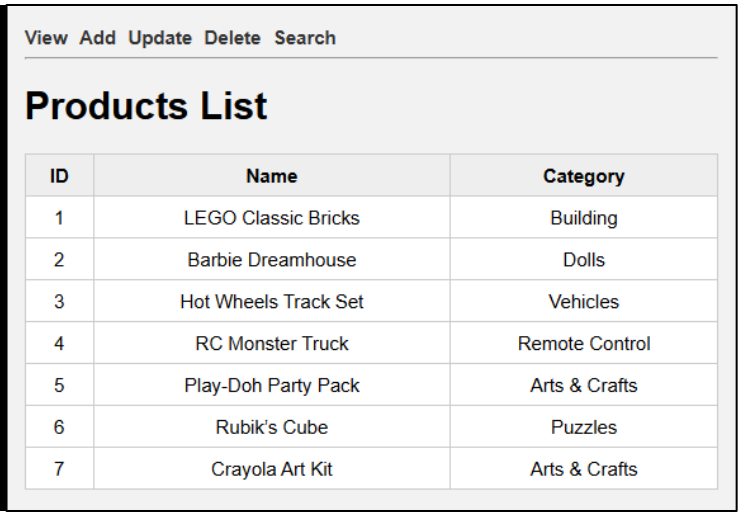
http://localhost:3000/products/
http://localhost:3000/products/add
http://localhost:3000/products/update
http://localhost:3000/products/delete
http://localhost:3000/products/search

Let's start by examining the simple front-end pages. Here is the common **PUG** header partial:

```
head
    meta(charset="UTF-8")
    title Product Management
    link(rel="stylesheet" href="/styles/style.css")
body
    nav
        a(href="/products/") View
        a(href="/products/add") Add
        a(href="/products/update") Update
        a(href="/products/delete") Delete
        a(href="/products/search") Search
        hr
```

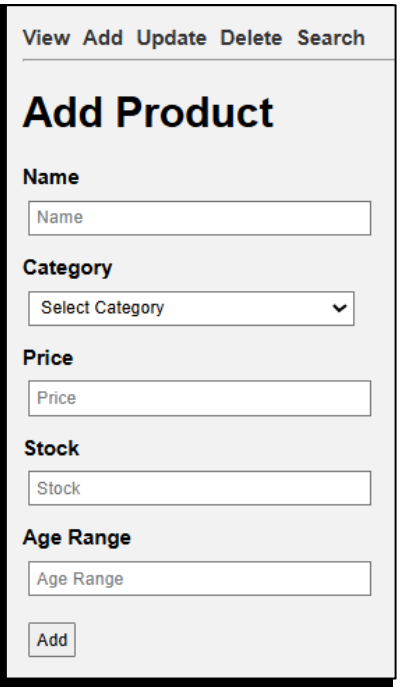
Here is the main page (i.e., **index.pug**) that uses this header and displays a list of products currently in the database:

```
doctype html
html
    include partials/header.pug

    h1 Products List
    table
        tr
            th ID
            th Name
            th Category
            each p in products
                tr
                    td= p._id
                    td= p.name
                    td= p.category
```

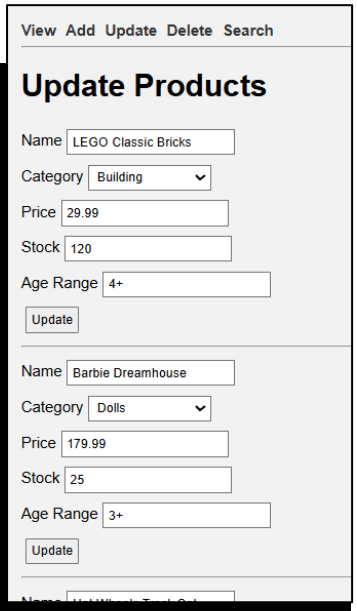


Here is what the **add.pug** will look like:

```
doctype html
html
    include partials/header.pug

    h1 Add Product
    form(action="/products/add" method="POST")
        div
            label(for="name") Name
            input#name(name="name" placeholder="Name" required)
        div
            label(for="category") Category
            select#category(name="category" required)
                option(value="" disabled selected) Select Category
                option(value="Building") Building
                option(value="Dolls") Dolls
                option(value="Vehicles") Vehicles
                option(value="Outdoor") Outdoor
                option(value="Arts & Crafts") Arts & Crafts
                option(value="Puzzles") Puzzles
                option(value="Games") Games
        div
            label(for="price") Price
            input#price(name="price" placeholder="Price" type="number" step="0.01" required)
        div
            label(for="stock") Stock
            input#stock(name="stock" placeholder="Stock" type="number" required)
        div
            label(for="ageRange") Age Range
            input#ageRange(name="ageRange" placeholder="Age Range" required)
        div
            button(type="submit") Add
```



Here is the **update.pug** file, which lists ALL products:




```
doctype html
html
    include partials/header.pug

    h1 Update Products

    each p in products
      form(action="/products/update" method="POST")
        div
            input(type="hidden" name="id" value=p._id)
        div
            label(for="name") Name
            input#name(name="name" value=p.name required)
        div
            label(for="category") Category
            select#category(name="category" required)
                option(value="" disabled) Select Category
                option(value="Building" selected=(p.category=="Building")) Building
                option(value="Dolls" selected=(p.category=="Dolls")) Dolls
                option(value="Vehicles" selected=(p.category=="Vehicles")) Vehicles
                option(value="Outdoor" selected=(p.category=="Outdoor")) Outdoor
                option(value="Arts & Crafts" selected=(p.category=="Arts & Crafts")) Arts & Crafts
                option(value="Puzzles" selected=(p.category=="Puzzles")) Puzzles
                option(value="Games" selected=(p.category=="Games")) Games
        div
            label(for="price") Price
            input#price(name="price" type="number" step="0.01" value=p.price required)
        div
            label(for="stock") Stock
            input#stock(name="stock" type="number" value=p.stock required)
        div
            label(for="ageRange") Age Range
            input#ageRange(name="ageRange" value=p.ageRange required)
        div
            button(type="submit") Update
        hr
```

Here is the **delete.pug** file, which lists ALL products as well:

```
doctype html
html
  include partials/header.pug

  h1 Delete Products

  table(border="1"
        cellpadding="8"
        cellspacing="0"
        style="border-collapse:collapse;")
    thead
      tr
        th ID
        th Name
        th Stock
        th Action
```

View Add Update Delete Search

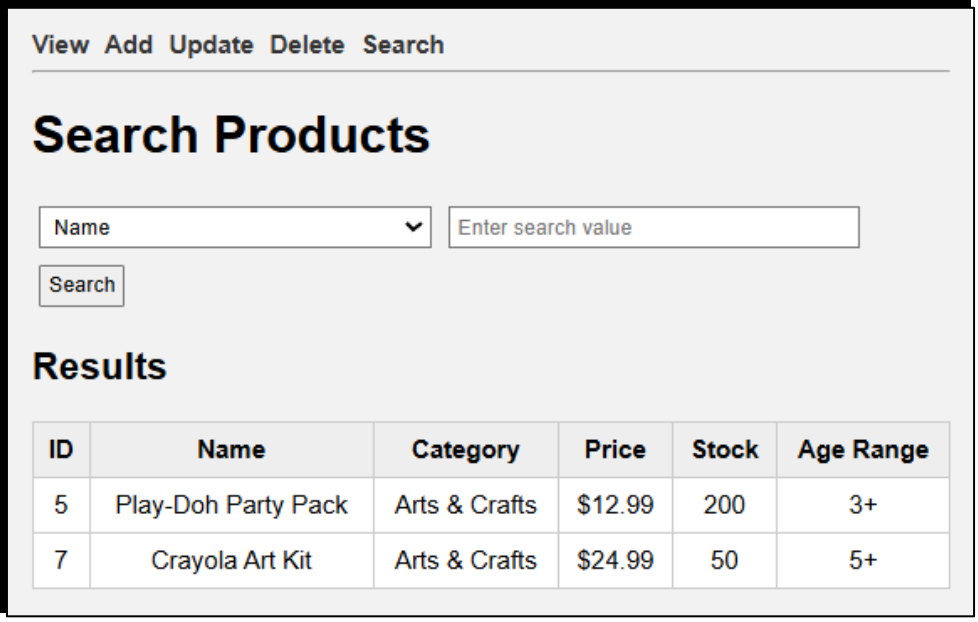
Delete Products

ID	Name	Stock	Action
1	LEGO Classic Bricks	120	Delete
2	Barbie Dreamhouse	25	Delete
3	Hot Wheels Track Set	75	Delete
4	RC Monster Truck	40	Delete
5	Play-Doh Party Pack	200	Delete
6	Rubik's Cube	150	Delete
7	Crayola Art Kit	50	Delete

```
    tbody
      each p in products
        tr
          td #{p._id}
          td #{p.name}
          td #{p.stock}
          td
            form(action="/products/delete" method="POST")
              input(type="hidden" name="id" value=p._id)
              button(type="submit") Delete
```

Finally, the **search.pug** allows to perform basic matching searches:

```
doctype html
html
    include partials/header.pug

    h1 Search Products
    form(action="/products/search" method="POST")
        select(name="field")
            option(value="name") Name
            option(value="category") Category
            option(value="ageRange") Age Range

        input(type="text"
              name="value"
              id="valueInput"
              placeholder="Enter search value" required)

        button Search

    if products.length
        h2 Results
        table
            tr
                th ID
                th Name
                th Category
                th Price
                th Stock
                th Age Range
            each p in products
                tr
                td= p._id
                td= p.name
                td= p.category
                td $#{p.price}
                td= p.stock
                td= p.ageRange
```



Of course, we can add a lot of functionality to these pages (especially the search page), but the idea of connecting to the database server is all we are trying to show here. Also, the **style.css** file is not described here but is fairly basic.

So, what does our server look like? Here is the **server.js** file. The "interesting stuff" is in the **products.js** router page which we will look at in a moment:

```
// Get the database module. Notice that we are never disconnecting
const { connectToDatabase } = require("./db");
const PORT = 3000;

// Get the Express object
const express = require("express");
const app = express();

// Set up Pug as the view engine
app.set("view engine", "pug");

// Middleware to parse HTML form data into req.body
app.use(express.urlencoded({ extended: true })); // true allows nested objects

// Tell Express where the styles and views are
app.use('/styles', express.static("styles"));

// Main function that starts the server
async function main() {
    const db = await connectToDatabase();
    const products = db.collection("products");

    // Mount products router
    app.use("/products", require("./routes/products")(products));

    // Home redirects to list products
    app.get("/", (req, res) => res.redirect("/products"));

    // Start server
    app.listen(PORT);
    console.log(`Server is listening at http://localhost:${PORT}`);
}

main();
```

Notice that we add middleware to indicate that we want to be able to parse the **HTML** form data query strings. The use of `{extended: true}` allows the parser to handle nested objects. We don't have any nested objects in our example, so we could use **false** here. However, if we wanted to use a similar template for our **FutureTech Corp.** site which has nested objects, then **true** would be needed. We also indicate to serve the **style.css** file as a static file. We do not have any other static files.

The code to connect to the database is in our **main()** function. We make sure to then handle the routing to the **/products/ API** in a different file and also allow a redirect of **http://localhost:3000/** to **http://localhost:3000/products/**.

Notice that we do not call `listen()` until AFTER we connect to the database. It is good to also notice that we do not do any error-checking here for connections to the database. That error-checking is in **db.js**, so we do not have to worry about it here.

So, what does our **products.js** look like? It does all the routing fun and communicates to the database. The structure of the code will look as shown below. The routing of the home page is handled in this code as well.

```
const express = require("express");
const router = express.Router();

module.exports = (products) => {

    // "View" products page
    router.get("/", async (req, res) => {
        const allProducts = await products.find().toArray();
        res.render("index", { products: allProducts });
    });

    // "Add" product page
    ...

    // "Update" product page
    ...

    // "Delete" product page
    ...

    // "Search" products page
    ...

    return router;
};
```

Previously, we simply did `module.exports = router;` to export our router. However, in the above code, we are exporting a function that takes a `products` parameter and this function returns the router. The use of a function here, allows use to pass in the database's `products` collection … which is required by the code. If we did not use a function here, the router itself would have to get access to the database collection internally by doing something like this in the router code:

```
let products; // Will hold the collection

// Immediately connect to DB when this module is loaded
(async () => {
    const db = await connectToDatabase();
    products = db.collection("products");
})();
```

In summary, by using the function with the `products` passed in, our code is cleaner, safer and easier to test and maintain than if we had to put the connection code in like this.

Now, notice how simple it is when someone requests the home page to view the products. We simply get all the products from the database and then render the **index.pug** page, making sure to pass in all the products into the **products** variable sent to the **PUG** page.

Now what about the **/add** page? It is also easy. We need to handle the **GET** for the **/add** page as well as the **POST** of the newly added item from the form. So, we need the two routes as follows:

```
// "Add" product page
router.get("/add", (req, res) => res.render("add"));

// Handle an add
router.post("/add", async (req, res) => {
    // Get the attributes from the request body
    const { name, category, price, stock, ageRange } = req.body;

    // Find the highest _id of all products
    const lastProduct = await products.findOne({}, { sort: { _id: -1 } });

    // Make sure to do an increment on the _id each time
    const nextId = lastProduct ? lastProduct._id + 1 : 1;

    // Now add the product
    await products.insertOne({
        _id: nextId,
        name,
        category,
        price: parseFloat(price),
        stock: parseInt(stock),
        ageRange
    });

    // Go back to the products view page
    res.redirect("/products");
});
```

Notice that rendering the **add.pug** page on a **GET** request is simple, since it needs no additional data. For the **POST** from the **/add** page, we first get the attributes from the request body. Then we look at all the products and find the largest `_id` so that we can choose an `_id` which is one higher (unless there are no products yet, in which case we choose an `_id` of **1**). Then all we need to do is call the `insertOne()` function.

Notice the use of `parseFloat()` and `parseInt()` to convert the form strings to numbers. We end by redirecting to the view page.

The handling of the **GET** and **POST** to the **/update** page is similar. The **GET** will list all products again and the **POST** will be simple in that it just requires calling `updateOne()`:

```
// "Update" product page
router.get("/update", async (req, res) => {
    const allProducts = await products.find().toArray();
    res.render("update", { products: allProducts });
});

// Handle an update
router.post("/update", async (req, res) => {
    const { id, name, category, price, stock, ageRange } = req.body;
```

```
    // Update the product now
    await products.updateOne(
        { _id: parseInt(id) },
        { $set: {name, category, price: parseFloat(price), stock: parseInt(stock), ageRange} }
    );

    // Go back to the products view page
    res.redirect("/products");
});
```

This code simply updates all the fields with whatever is currently in the form. However, when calling `updateOne()` we only need to pass in the fields that have changed. But this would take more work. We would either need to get the current item again (i.e., the one with the same `_id`) and compare the new values to see which ones changed, or we can use **JavaScript** on the client side to keep track of which fields the user changed. The first approach would look something like this:

```
// Handle an update
router.post("/update", async (req, res) => {
    const { id, name, category, price, stock, ageRange } = req.body;

    // Get the current product with the same _id
    const current = await products.findOne({ _id: parseInt(id) });
    const updates = {};

    if (current.name !== name) updates.name = name;
    if (current.category !== category) updates.category = category;
    if (current.price !== parseFloat(price)) updates.price = parseFloat(price);
    if (current.stock !== parseInt(stock)) updates.stock = parseInt(stock);
    if (current.ageRange !== ageRange) updates.ageRange = ageRange;

    if (Object.keys(updates).length > 0) {
        await products.updateOne({ _id: parseInt(id) }, { $set: updates });
    }

    // Go back to the products view page
    res.redirect("/products");
});
```

This coding approach will work as long as the **"id"** form field on our rendered **HTML** page (that the user interacts with) is visibly **"hidden"**:

```
input(type="hidden" name="id" value=p._id)
```

By hiding this input field, the user cannot alter it, thereby ensuring that we are looking up the same `_id` as the product being updated because the user cannot change it to some other `_id`, which would cause problems.

The **/delete** endpoint is also quite similar to handle, yet it is even simpler since we only need the `_id` field:

```
// "Delete" product page
router.get("/delete", async (req, res) => {
    const allProducts = await products.find().toArray();
    res.render("delete", { products: allProducts });
});

// Handle a delete
router.post("/delete", async (req, res) => {
    const { id } = req.body;
    await products.deleteOne({ _id: parseInt(id) });

    // Go back to the products view page
    res.redirect("/products");
});
```

Likewise, handling the **/search** page is also simple:

```
// "Search" products page
router.get("/search", (req, res) => res.render("search", { products: [] }));

// Handle a search
router.post("/search", async (req, res) => {
    const { field, value } = req.body;
    const query = {};
    if (field && value)
         query[field] = value;

    const results = await products.find(query).toArray();
    res.render("search", { products: results });
});
```

Notice that for a **GET**, we start off with no data (i.e., no products) so we do not need to access the database … we just pass in an empty list of products to render on the **search.pug** page. Otherwise, we look at the field and its value to search for and set it in our query variable (which is initially an empty object). Then we just call `find()` for the results and render them. We do not re-direct this time, since we want to see the results appear on this page.

As we have seen in this chapter, by connecting a **PUG Express** server to a **MongoDB** database, we open up a whole new world of data for our website. Instead of just showing fixed content, our site can store and retrieve information, let users add new items, update existing ones, delete old ones, and search through all the data in real time.

Once we become comfortable with this kind of setup, it's almost like giving our website a memory: it can remember user actions, respond to changes, and adjust over time. From here, we can start experimenting, add more advanced features, and really explore the power of working with live, dynamic data.