Part-B

1.b) Delete Database

Syntax:

db.dropDatabase()

Example:

>use mydb

switched to db mydb

>db.dropDatabase()

>{ “dropped” : “mydb” , “ok” : 1 }

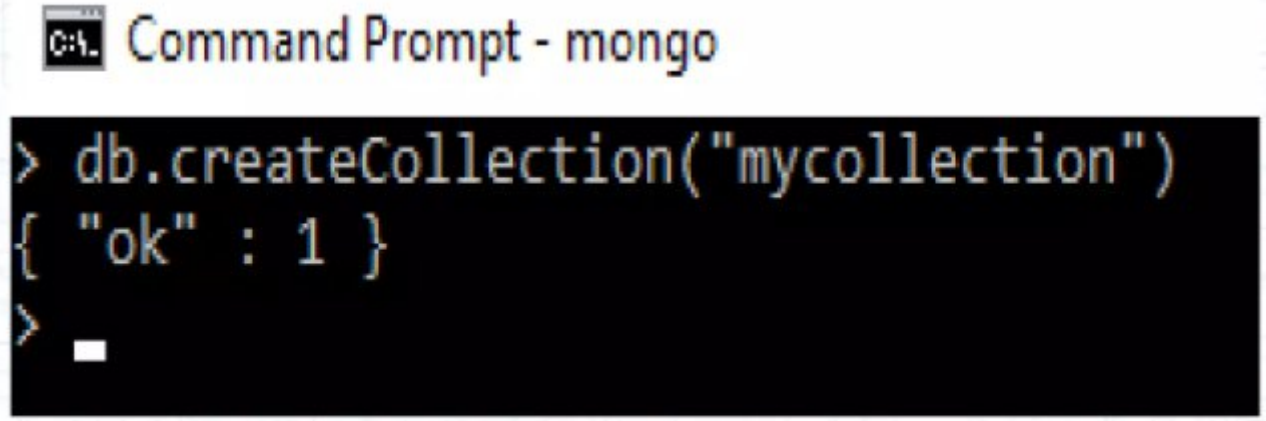
>

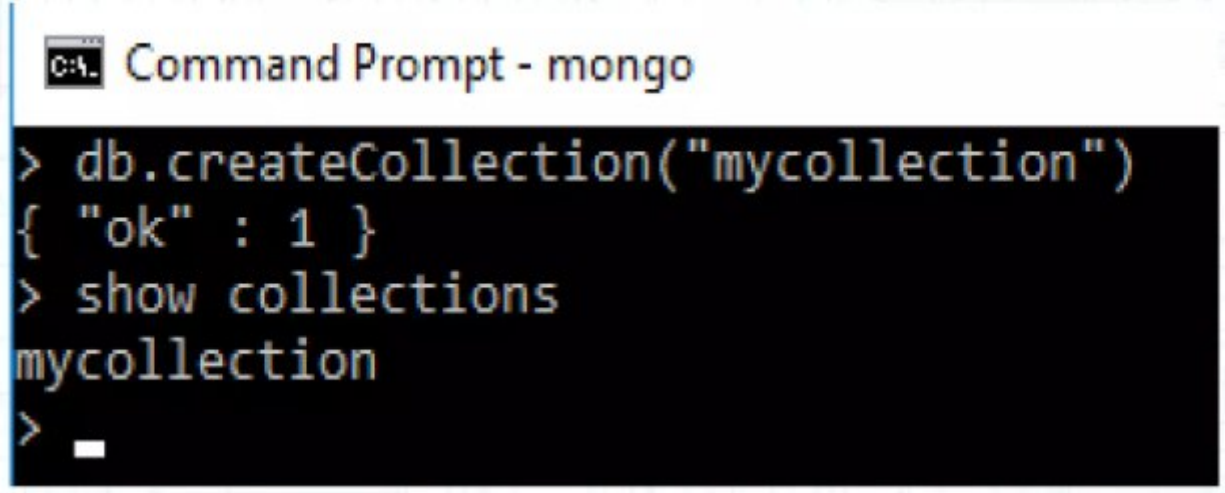
2.a) Create Collection

Syntax:

db.createCollection(name, options)

Parameter	Type	Description
name	String	Name of the collection to be created
options	Document	(optional) specify options about memory size and indexing


Command Prompt - mongo
> db.createCollection("mycollection")
{ "ok" : 1 }
>


Command Prompt - mongo
> db.createCollection("mycollection")
{ "ok" : 1 }
> show collections
mycollection
>

2.b) Drop collection

Syntax:

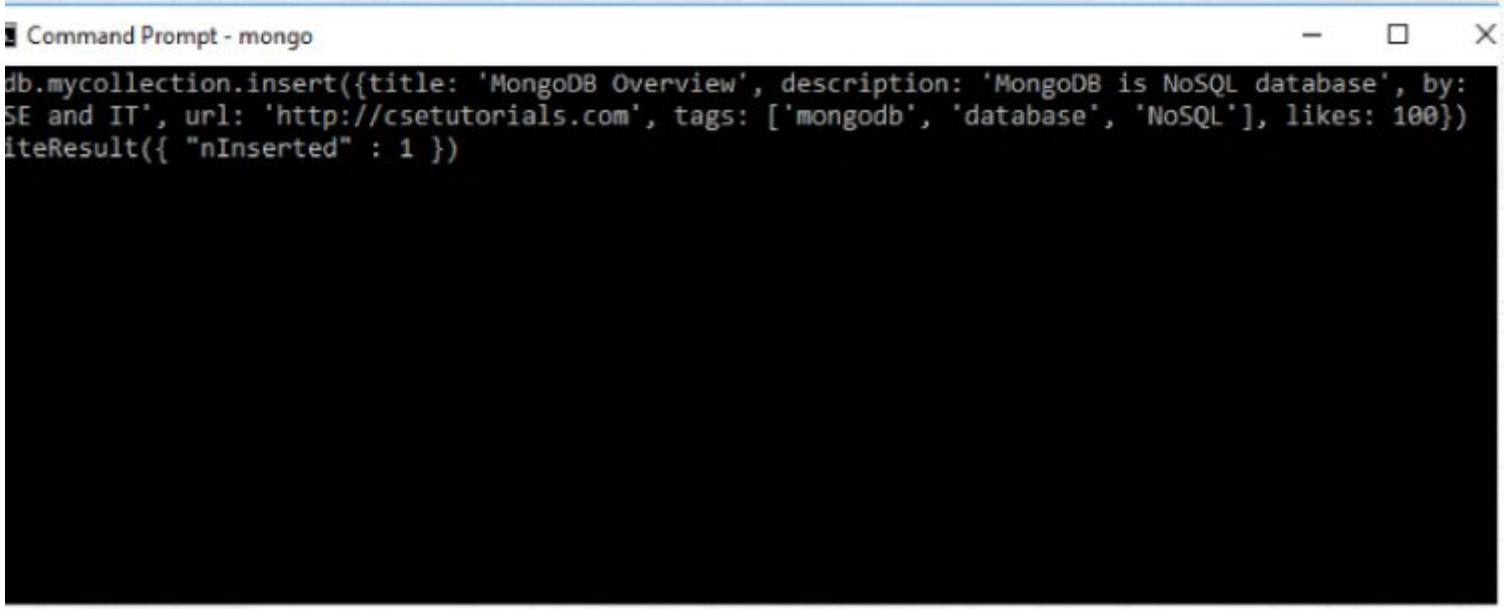
db.collection_name.drop()



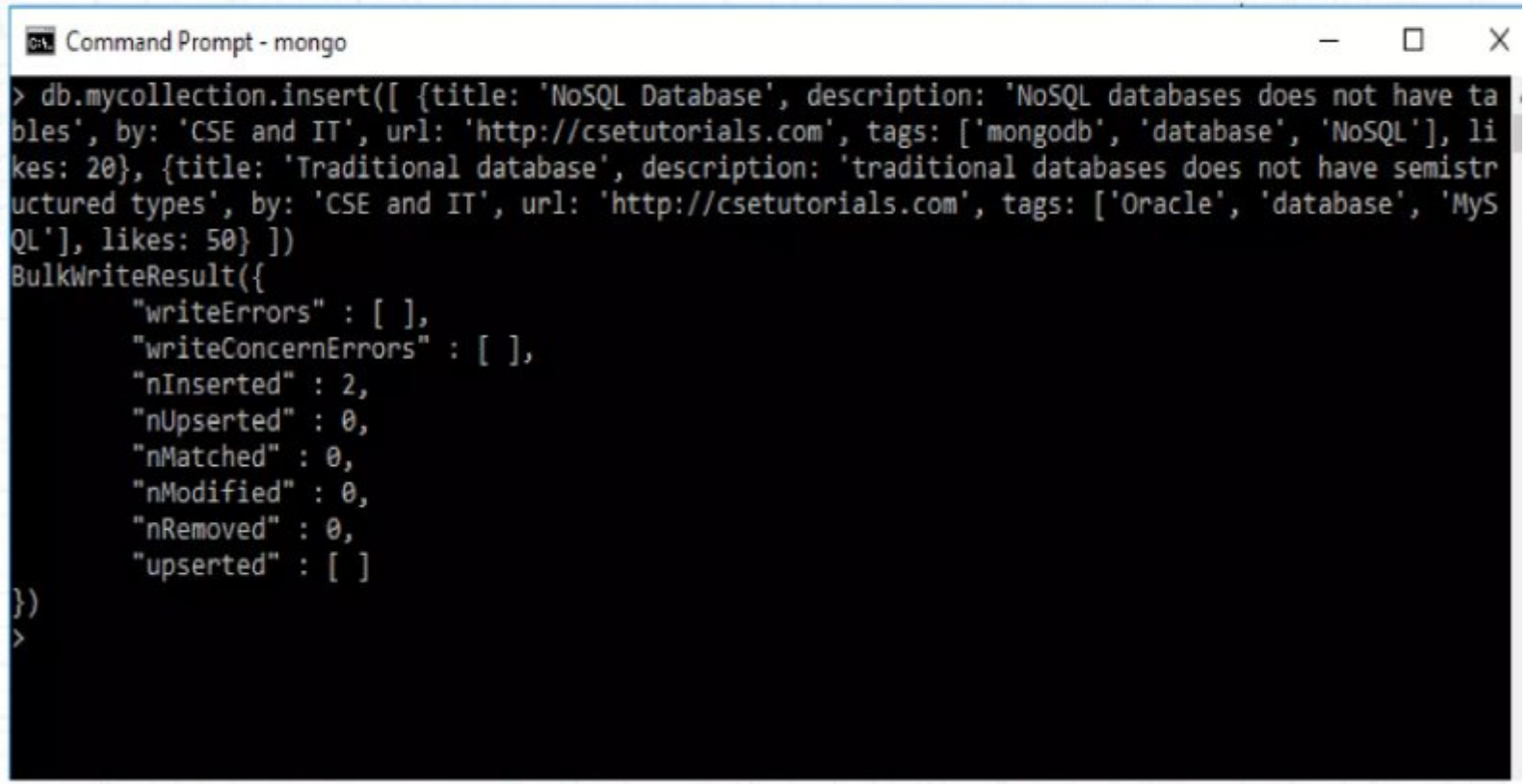
3.a) Insert document

Syntax:

db.collection_name.insert(document)



We can store multiple documents also at the same time using insert() command like the following.



3.b) Query document

To query from MongoDB collection, we need to use find() method.

Syntax:

```
db.collection_name.find()
```

This will display all the documents in a non-structured way.

To display the results in a formatted way, pretty method is used.

Syntax:

```
db.collection_name.find().pretty()
```

To return only one document, we have findOne() method.

Syntax:

```
db.collection_name.findOne()
```

This will display the first document in the given collection.

3.c) Update document

The update() method is used to update the values in the existing document.

Syntax:

```
db.collection_name.update(selection criteria,
                          updated data)
```

In the given example, the title is changed from 'MongoDB Overview' to 'New MongoDB Tutorial' for title field.

By default, MongoDB will update only a single document. To update multiple documents, we need to set a parameter 'multi' to true.

3.d) Remove document

remove() method is used to remove a document from the collection.

Syntax:

```
db.collection_name.remove(deletion_criteria)
```

We can use deletion_criteria or justOne. Both the values usage are optional only.

All the documents satisfying the given criteria will be deleted.

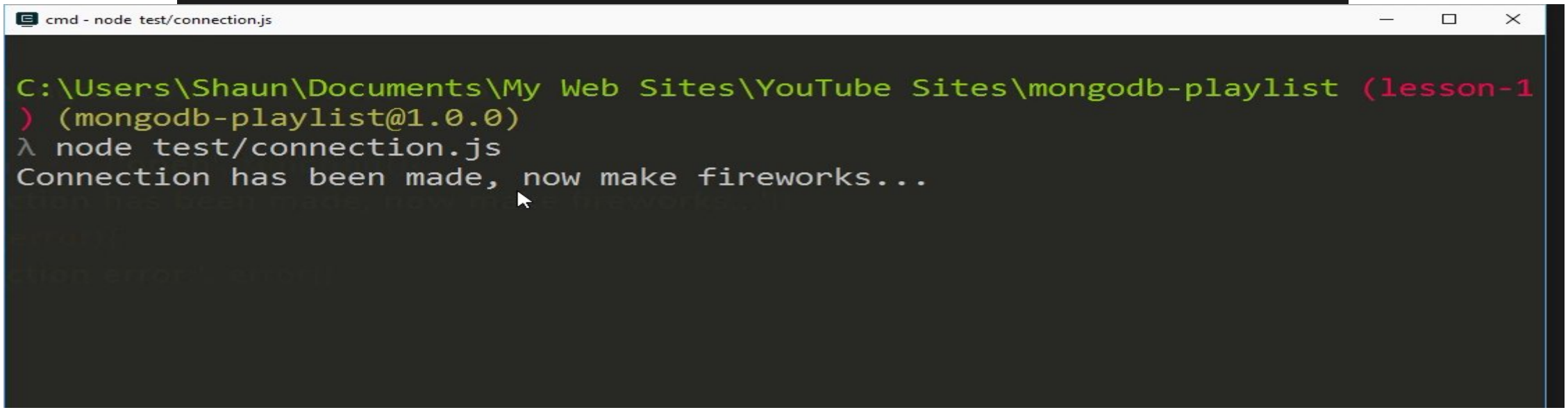
Online execution of queries

https://www.jdoodle.com/online-mongodb-terminal

```
const mongoose = require('mongoose');

// Connect to mongodb
mongoose.connect('mongodb://localhost/testaroo');

mongoose.connection.once('open', function(){
   console.log('Connection has been made, now make fireworks...');
}).on('error', function(error){
   console.log('Connection error:', error);
});
```



Data Types in MongoDB

In **MongoDB**, the documents are stored in **BSON** which is the binary encoded format of **JSON** and using BSON we can make remote procedure calls in MongoDB. BSON data format supports various data types.

String:

This is the most commonly used data type in MongoDB to store data, BSON strings are of **UTF-8.** So, the **drivers** for each **programming language** convert from data types to the string format of the language to **UTF-8**

```
switched to db gfg
> db.student.insertMany([{name:"Akshay"},{name:"Vikash"}])
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("601af2dd6fd54aa34c9c6df3"),
                ObjectId("601af2dd6fd54aa34c9c6df4")
        ]
}
> db.student.find().pretty()
{ "_id" : ObjectId("601af2dd6fd54aa34c9c6df3"), "name" : "Akshay" }
{ "_id" : ObjectId("601af2dd6fd54aa34c9c6df4"), "name" : "Vikash" }
>
```

Integer

store integer data type in two forms 32-bit signed integer and 64-bit signed integer.

```
> db.student.insertOne({name:"Akash",age:19})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("601af3456fd54aa34c9c6df5")
}
> db.student.find().pretty()
{ "_id" : ObjectId("601af2dd6fd54aa34c9c6df3"), "name" : "Akshay" }
{ "_id" : ObjectId("601af2dd6fd54aa34c9c6df4"), "name" : "Vikash" }
{
        "_id" : ObjectId("601af3456fd54aa34c9c6df5"),
        "name" : "Akash",
        "age" : 19
}
>
```

Double

The **double** data type is used for storing floating-point numbers

```
}
> db.student.insertOne({name:"Sagar",age:20,marks:546.43})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("601af4126fd54aa34c9c6df6")
}
> db.student.find().pretty()
{ "_id" : ObjectId("601af2dd6fd54aa34c9c6df3"), "name" : "Akshay" }
{ "_id" : ObjectId("601af2dd6fd54aa34c9c6df4"), "name" : "Vikash" }
{
        "_id" : ObjectId("601af3456fd54aa34c9c6df5"),
        "name" : "Akash",
        "age" : 19
}
{
        "_id" : ObjectId("601af4126fd54aa34c9c6df6"),
        "name" : "Sagar",
        "age" : 20,
        "marks" : 546.43
}
>
```

Boolean

The **boolean** data type stores one of two values: **true** or **false**

```
> db.student.insertOne({name:"vishal",pass:true})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("601af47d6fd54aa34c9c6df7")
}
> db.student.find().pretty()
{ "_id" : ObjectId("601af2dd6fd54aa34c9c6df3"), "name" : "Akshay" }
{ "_id" : ObjectId("601af2dd6fd54aa34c9c6df4"), "name" : "Vikash" }
{
        "_id" : ObjectId("601af3456fd54aa34c9c6df5"),
        "name" : "Akash",
        "age" : 19
}
{
        "_id" : ObjectId("601af4126fd54aa34c9c6df6"),
        "name" : "Sagar",
        "age" : 20,
        "marks" : 546.43
}
{
        "_id" : ObjectId("601af47d6fd54aa34c9c6df7"),
        "name" : "vishal",
        "pass" : true
}
>
```

Null

The **null** data type stores a **null** value. This is useful when you want to represent the absence of data, such as an optional field that may not be set.

```
> db.student.insertOne({name:"Akash",MobNo:null,skills:["c","c++","java","python","JS"]})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("601af5fb6fd54aa34c9c6df8")
}
> db.student.find().pretty()
{ "_id" : ObjectId("601af2dd6fd54aa34c9c6df3"), "name" : "Akshay" }
{ "_id" : ObjectId("601af2dd6fd54aa34c9c6df4"), "name" : "Vikash" }
{
        "_id" : ObjectId("601af3456fd54aa34c9c6df5"),
        "name" : "Akash",
        "age" : 19
}
{
        "_id" : ObjectId("601af4126fd54aa34c9c6df6"),
        "name" : "Sagar",
        "age" : 20,
        "marks" : 546.43
}
{
        "_id" : ObjectId("601af47d6fd54aa34c9c6df7"),
        "name" : "vishal",
        "pass" : true
}
{
        "_id" : ObjectId("601af5fb6fd54aa34c9c6df8"),
        "name" : "Akash",
        "MobNo" : null,
        "skills" : [
                "c",
                "c++",
                "java",
                "python",
                "JS"
        ]
}
>
```

```
> db.student.insertOne({name:"Akash",MobNo:null,skills:["c","c++","java","python","JS"]})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("601af5fb6fd54aa34c9c6df8")
}
> db.student.find().pretty()
{ "_id" : ObjectId("601af2dd6fd54aa34c9c6df3"), "name" : "Akshay" }
{ "_id" : ObjectId("601af2dd6fd54aa34c9c6df4"), "name" : "Vikash" }
{
        "_id" : ObjectId("601af3456fd54aa34c9c6df5"),
        "name" : "Akash",
        "age" : 19
}
{
        "_id" : ObjectId("601af4126fd54aa34c9c6df6"),
        "name" : "Sagar",
        "age" : 20,
        "marks" : 546.43
}
{
        "_id" : ObjectId("601af47d6fd54aa34c9c6df7"),
        "name" : "vishal",
        "pass" : true
}
{
        "_id" : ObjectId("601af5fb6fd54aa34c9c6df8"),
        "name" : "Akash",
        "MobNo" : null,
        "skills" : [
                "c",
                "c++",
                "java",
                "python",
                "JS"
        ]
}
>
```

Array

The **array** data type allows us to store multiple values in a single field.
MongoDB arrays can contain values of the same or different data types

```
> db.student.find().pretty()
{ "_id" : ObjectId("601af2dd6fd54aa34c9c6df3"), "name" : "Akshay" }
{ "_id" : ObjectId("601af2dd6fd54aa34c9c6df4"), "name" : "Vikash" }
{
        "_id" : ObjectId("601af3456fd54aa34c9c6df5"),
        "name" : "Akash",
        "age" : 19
}
{
        "_id" : ObjectId("601af4126fd54aa34c9c6df6"),
        "name" : "Sagar",
        "age" : 20,
        "marks" : 546.43
}
{
        "_id" : ObjectId("601af47d6fd54aa34c9c6df7"),
        "name" : "vishal",
        "pass" : true
}
{
        "_id" : ObjectId("601af5fb6fd54aa34c9c6df8"),
        "name" : "Akash",
        "MobNo" : null,
        "skills" : [
                "c",
                "c++",
                "java",
                "python",
                "JS"
        ]
}
>
```

Object (Embedded Document)

Object data type stores embedded documents. Embedded documents are also known as nested documents.

```
> db.book.insertOne({Book:{name:"C in depth",writer:"Aaksh"}})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("601af71f6fd54aa34c9c6df9")
}
> db.bool.find().pretty()
> db.book.find().pretty()
{
        "_id" : ObjectId("601af71f6fd54aa34c9c6df9"),
        "Book" : {
                "name" : "C in depth",
                "writer" : "Aaksh"
        }
}
>
```

Object Id

Whenever we create a new document in the collection MongoDB automatically creates a unique object id for that document
The data which is stored in Id is of hexadecimal format and the length of the id is 12 bytes which consist:

•4-bytes for Timestamp value.

•5-bytes for Random values. i.e., 3-bytes for machine Id and

•2-bytes for process Id.

•3- bytes for Counter

Undefined

The **undefined** data type represents a value that is not defined. It is rarely used in modern MongoDB applications, as it has been replaced by the null value for most practical purposes.

```
> db.project.insertOne({name:"FauG",duration:undefined,binaryValue:"110011001"})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("601af8516fd54aa34c9c6dfa")
}
> db.project.find().pretty()
{
        "_id" : ObjectId("601af8516fd54aa34c9c6dfa"),
        "name" : "FauG",
        "duration" : undefined,
        "binaryValue" : "110011001"
}
>
```

Binary Data

The **binary data** data type stores binary information such as images, files, or encrypted data. Binary data is often used when working with non-textual data.

```
> db.project.insertOne({name:"FauG",duration:undefined,binaryValue:"110011001"})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("601af8516fd54aa34c9c6dfa")
}
> db.project.find().pretty()
{
        "_id" : ObjectId("601af8516fd54aa34c9c6dfa"),
        "name" : "FauG",
        "duration" : undefined,
        "binaryValue" : "110011001"
}
```

Date

Date data type stores date. It is a 64-bit integer which represents the number of milliseconds.
BSON data type generally supports UTC datetime and it is signed. If the value of the date data type is negative then it represents the dates before 1970.
There are various methods to return date, it can be returned either as a string or as a date object. Some method for the date:

- **Date():** It returns the current date in string format.
- **new Date():** Returns a date object. Uses the ISODate() wrapper.
- **new ISODate():** It also returns a date object. Uses the ISODate() wrapper.

```
>
> var date1  = Date()
> var date2 = new Date()
> var date3 = new ISODate()
> db.date.insertOne({Date1:date1,Date2:date2,Date3:date3})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("601afa326fd54aa34c9c6dfc")
}
> db.date.find().pretty()
{
        "_id" : ObjectId("601afa326fd54aa34c9c6dfc"),
        "Date1" : "Thu Feb 04 2021 01:00:34 GMT+0530 (India Standard Time)",
        "Date2" : ISODate("2021-02-03T19:30:40.014Z"),
        "Date3" : ISODate("2021-02-03T19:30:56.454Z")
}
```

Min & Max key

The **MinKey** and **MaxKey** data types are used for internal comparisons.
MinKey represents the lowest BSON element, while **MaxKey** represents the highest.

```
>
> db.m.insertOne({min:MinKey,max:MaxKey})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("601afbd66fd54aa34c9c6dff")
}
> db.m.find().pretty()
{
        "_id" : ObjectId("601afbd66fd54aa34c9c6dff"),
        "min" : { "$minKey" : 1 },
        "max" : { "$maxKey" : 1 }
}
>
```

Symbol

This data type similar to the string data type. It is generally not supported by a mongo shell, but if the shell gets a symbol from the database, then it converts this type into a string type.

```
}
> var symbl = "nik432#"
> db.symbl.insert({symbol:symbl})
WriteResult({ "nInserted" : 1 })
> db.symbl.find().pretty()
{ "_id" : ObjectId("601afb6f6fd54aa34c9c6dfe"), "symbol" : "nik432#" }
>
```

Regular Expression

The **regular expression** data type is used to store regex patterns.
MongoDB supports regex queries for performing pattern matching on string fields.

```
        binaryValue  :  11001001
}
> var expression = new RegExp("%gfg")
> db.exp.insertOne({Regular Expression: expression})
uncaught exception: SyntaxError: missing : after property id
@(shell):1:26
> db.exp.insertOne({RegularExpression: expression})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("601af91f6fd54aa34c9c6dfb")
}
> db.exp.find().pretty()
{
        "_id" : ObjectId("601af91f6fd54aa34c9c6dfb"),
        "RegularExpression" : /%gfg/
}
>
```

JavaScript

MongoDB allows us to store **JavaScript** code within documents using the **JavaScript** data type. This can be useful for storing code or expressions.

```
>
> var date1  = Date()
> var date2 = new Date()
> var date3 = new ISODate()
> db.date.insertOne({Date1:date1,Date2:date2,Date3:date3})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("601afa326fd54aa34c9c6dfc")
}
> db.date.find().pretty()
{
        "_id" : ObjectId("601afa326fd54aa34c9c6dfc"),
        "Date1" : "Thu Feb 04 2021 01:00:34 GMT+0530 (India Standard Time)",
        "Date2" : ISODate("2021-02-03T19:30:40.014Z"),
        "Date3" : ISODate("2021-02-03T19:30:56.454Z")
}
>
```

Timestamp

In MongoDB, this data type is used to store a timestamp. It is useful when we modify our data to keep a record and the value of this data type is 64-bit.
The value of the timestamp data type is always unique.

```
> var timestmp = new Timestamp()
> timestmp
Timestamp(0, 0)
> db.time.insertOne({Timestamp: timestmp})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("601afafc6fd54aa34c9c6dfd")
}
> db.time.find().pretty()
{
        "_id" : ObjectId("601afafc6fd54aa34c9c6dfd"),
        "Timestamp" : Timestamp(1612380924, 1)
}
>
```

Decimal

This MongoDB data type store 128-bit decimal-based floating-point value.
This data type was introduced in MongoDB version 3.4

```
> var decimal = NumberDecimal("1000.55")
> decimal
NumberDecimal("1000.55")
> db.decimal.insertOne({NumberDecimal: decimal})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("6299c81c2fe2f881ac89c8a6")
}
> db.decimal.find().pretty()
{
        "_id" : ObjectId("6299c81c2fe2f881ac89c8a6"),
        "NumberDecimal" : NumberDecimal("1000.55")
}
>
>
```

2.Insert multiple documents in collection.

To insert multiple documents we can use insertMany() - which takes an array of documents as parameter. When executed, it returns multiple ids for each document in the array. To drop the collection, use drop() command. Sometimes, when doing bulk inserts - we may insert duplicate values. Specifically, if we try to insert duplicate _ids, we'll get the duplicate key error:

Creating new collection:

>db.create("startup")

{ ok: 1 }

inserting many documents

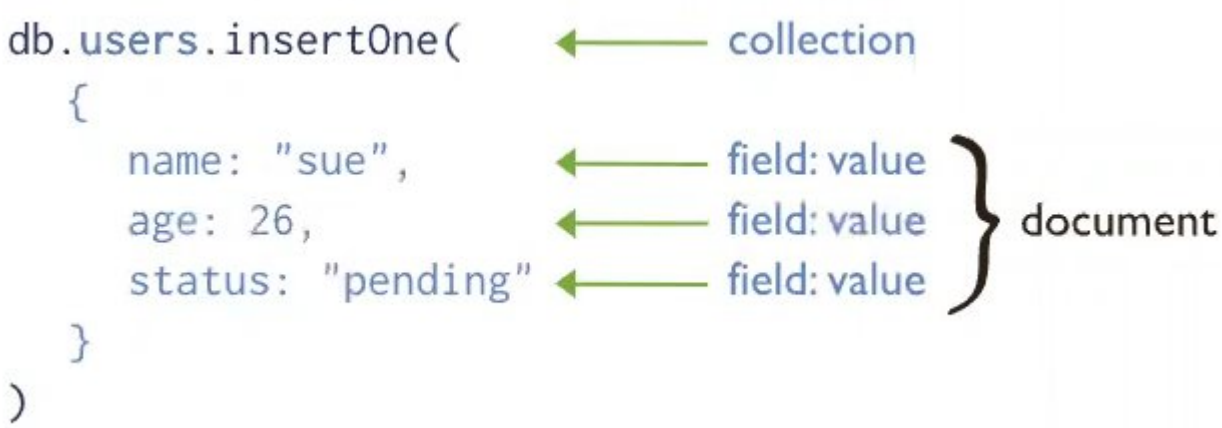
>db.startup.insertMany([

{_id:"id1", name:"Uber"},

{_id:"id2", name:"Airbnb"},

{_id:"id4", name:"Ola"},

]);



Operation	Syntax	Example	RDBMS Equivalent
Equality	{<key>:<value>}	db.mycol.find({"by":"tutorials point"}).pretty()	where by = 'tutorials point'
Less Than	{<key>:{$lt:<value>}}	db.mycol.find({"likes":{$lt:50}}).pretty()	where likes < 50
Less Than Equals	{<key>:{$lte:<value>}}	db.mycol.find({"likes":{$lte:50}}).pretty()	where likes <= 50
Greater Than	{<key>:{$gt:<value>}}	db.mycol.find({"likes":{$gt:50}}).pretty()	where likes > 50
Greater Than Equals	{<key>:{$gte:<value>}}	db.mycol.find({"likes":{$gte:50}}).pretty()	where likes >= 50
Not Equals	{<key>:{$ne:<value>}}	db.mycol.find({"likes":{$ne:50}}).pretty()	where likes != 50

Comparison Selectors: Comparison selectors are used to compare fields against specific values or other fields. Here are some common comparison selectors:

$eq - Matches values that are equal to a specified value.

$ne - Matches all values that are not equal to a specified value.

$gt - Matches values that are greater than a specified value.

$gte - Matches values that are greater than or equal to a specified value.

$lt - Matches values that are less than a specified value.

$lte - Matches values that are less than or equal to a specified value.

$in - Matches any of the values specified in an array.

$nin - Matches none of the values specified in an array.

Logical Selectors: Logical selectors are used to combine multiple conditions in a query. Here are some common logical selectors:

$and - Joins query clauses with a logical AND and requires that all conditions be true.

$or - Joins query clauses with a logical OR and requires that at least one condition be true.

$not - Inverts the effect of a query expression and returns documents that do not match the query expression.

$nor - Joins query clauses with a logical NOR and requires that none of the conditions be true.

```
>db.collection.find({
   $and: [
      { age: { $gte: 18 } },          // age greater than or equal to 18
      { age: { $lt: 30 } },           // age less than 30
      { gender: "male" },             // gender equals "male"
      { city: { $in: ["New York", "Los Angeles"] } }   // city is either "New York" or "Los Angeles"
   ]
})
```

Geospatial Selectors: MongoDB supports geospatial queries for geospatial data. It provides two types of geospatial indexes: 2d indexes and 2d sphere indexes.

Example: Let's say we have a collection named "locations" with documents containing location data

```
{
  "_id": ObjectId("60cf9df572f71e01c86bf703"),
  "name": "Store A",
  "location": {  "type": "Point",   "coordinates": [40.7128, -74.0060] }
},
{
  "_id": ObjectId("60cf9df572f71e01c86bf704"),
  "name": "Store B",
  "location": {"type": "Point",  "coordinates": [34.0522, -118.2437]  }
},
{
  "_id": ObjectId("60cf9df572f71e01c86bf705"),
  "name": "Store C",
  "location": { "type": "Point",  "coordinates": [51.5074, -0.1278]  }
}
```

To perform a geospatial query to find stores within a certain radius of a given location, you can use the $near operator:

```
>db.locations.find({
  location: {
    $near: {
      $geometry: {
        type: "Point",
        coordinates: [40.7128, -74.0060] // Coordinates of New York City
      },
      $maxDistance: 10000 // Maximum distance in meters (10 km)
    }
  }
})
```

$elemMatch: The $elemMatch operator is used to match documents that contain an array field with a least one element that matches all the specified query criteria.

$slice: The $slice projection operator is used within the projection document to limit the number of elements returned from an array field.

```
db.student2.insert([
  { "_id": 1, "name": "Alice", "grades": [85, 90, 78] },
  { "_id": 2, "name": "Bob", "grades": [95, 88, 82] },
  { "_id": 3, "name": "Charlie", "grades": [72, 92, 80] },
  { "_id": 4, "name": "David", "grades": [80, 85, 90] },
  { "_id": 5, "name": "Emily", "grades": [75, 87, 89] }
]
)
```

```
>db.student.find( {}, { "grades": { $slice: 2 } } )
>db.student.find( {}, { "grades": { $slice: -3 } } )

// To project only the first element in the grades array that is greater than or equal to 85, we can use the following query:
>db.student.find( { grades: { $gte: 85 } }, { "grades.$": 1 } )

//grades array contains at least one element that is greater than or equal to 85 and less than or equal to 90.
>db.students.find( { grades: { $elemMatch: { $gte: 85, $lte: 90 } } }, { name: 1, grades: 1 } )
```

Execute Aggregation operations ($avg, $min, $max, $push, $addToSet etc.) Encourage students to execute several queries to demonstrate various aggregation operators.

$avg: Calculates the average value of the numeric values in a specified array.

Example: Suppose we have a collection named "sales" with documents containing sales data:

```
{
  "_id": ObjectId("60cf9df572f71e01c86bf703"),
  "product": "A",
  "sales": [100, 150, 200]
},
{
  "_id": ObjectId("60cf9df572f71e01c86bf704"),
  "product": "B",
  "sales": [120, 130, 140]
}
>db.sales.aggregate([ { $project: { product: 1, averageSales: { $avg: "$sales" } } } ])
```

Connecting MongoDB to Node.js with Mongoose

- Mongoose is an **ODM(Object Data Modeling)** tool that helps to define the model based on the schema.
- A schema is a kind of structure that defines how we can store data in the database. It helps us to validate types like objects, strings, booleans, numbers, etc.

Mongoose is a library provided by NPM (Node Package Manager) for the MongoDB database.

With it, you can define collection schema, relationships between collections, database query builders, lookups, aggregation pipelines, etc.

Adding the MongoDB Driver to Node.js

Prerequisites

- **Node.js and npm**
- Node.js is a JavaScript runtime for building fast and scalable network applications, and it comes with npm installed. You can download an installer for your platform from the Node.js website.
- **MongoDB Node.js driver**
- To connect your application with a database, you need a software component, also known as a driver. MongoDB Node.js driver supports database connection, authentication, CRUD operations, and observations, among other features.
- **MongoDB Atlas**
- You can create a database either locally or in the cloud by using MongoDB Atlas, a fully managed cloud database service.

1. Install NodeJs
2. Instal NPM
3. Instal express server
4. Instal MongoDB
5. Run MingoDb database
    1. Mongod
    2. Create StudentDB database
    3. Create Students collection

```
student-app/
├── student-backend/
│   ├── server.js        ← Modern backend
│   ├── package.json
│   └── node_modules/
└── student-frontend/
    ├── src/App.js
    ├── package.json
    └── node_modules/
```

1.Start MongoDB (`mongod.exe --dbpath="C:\data\db"`)
2.Start Backend (`node server.js`)
3.Start Frontend (`npm start`)
React fetches data from Express → Express fetches from MongoDB → Display on frontend.

Step 1: Prepare Your MongoDB Database

1 Start MongoDB

For Local MongoDB:

1.Open **Command Prompt as Administrator**.

2.Make sure your data folder exists:

```
mkdir C:\data\db
```

3. Start MongoDB:

```
"C:\Program Files\MongoDB\Server\8.2\bin\mongod.exe" --dbpath="C:\data\db"
```

•Keep this terminal **open**.

•MongoDB default port: 27017.

Step 2: Setup Backend (Node.js + Express + Mongoose)

2 Create a backend folder

student-app/student-backend/

3. Initialize Node.js

```
cd "C:\My project\pvp\Subjects\2025-26\Sem-1\MERN\React Programs\student-app\student-backend"
npm init -y
```

4.Install dependencies

```
npm install express mongoose cors
```

5.Create `server.js`

```
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');

const app = express();
app.use(cors());
app.use(express.json());

// Connect to MongoDB
mongoose.connect('mongodb://127.0.0.1:27017/studentDB')
  .then(() => console.log('✅ Connected to MongoDB'))
  .catch(err => console.error('❌ MongoDB connection error:', err.message));

// Flexible schema
const studentSchema = new mongoose.Schema({}, { strict: false });
const Student = mongoose.model('Student', studentSchema, 'students');
```

- strict: false → allows the schema to adapt to any existing fields in your MongoDB collection.
- 'students' → explicitly tells Mongoose to use the existing students collection.

```
// Routes
app.get('/students', async (req, res) => {
  try {
    const students = await Student.find();
    res.json(students);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});

app.post('/students', async (req, res) => {
  try {
    const newStudent = new Student(req.body); // Create a new document
    const savedStudent = await newStudent.save(); // Save to MongoDB
    res.status(201).json(savedStudent); / Return saved student
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
});

        // Start server
        const PORT = 5000;
        app.listen(PORT, () => console.log(` Server running on
        http://localhost:${PORT}`));
```

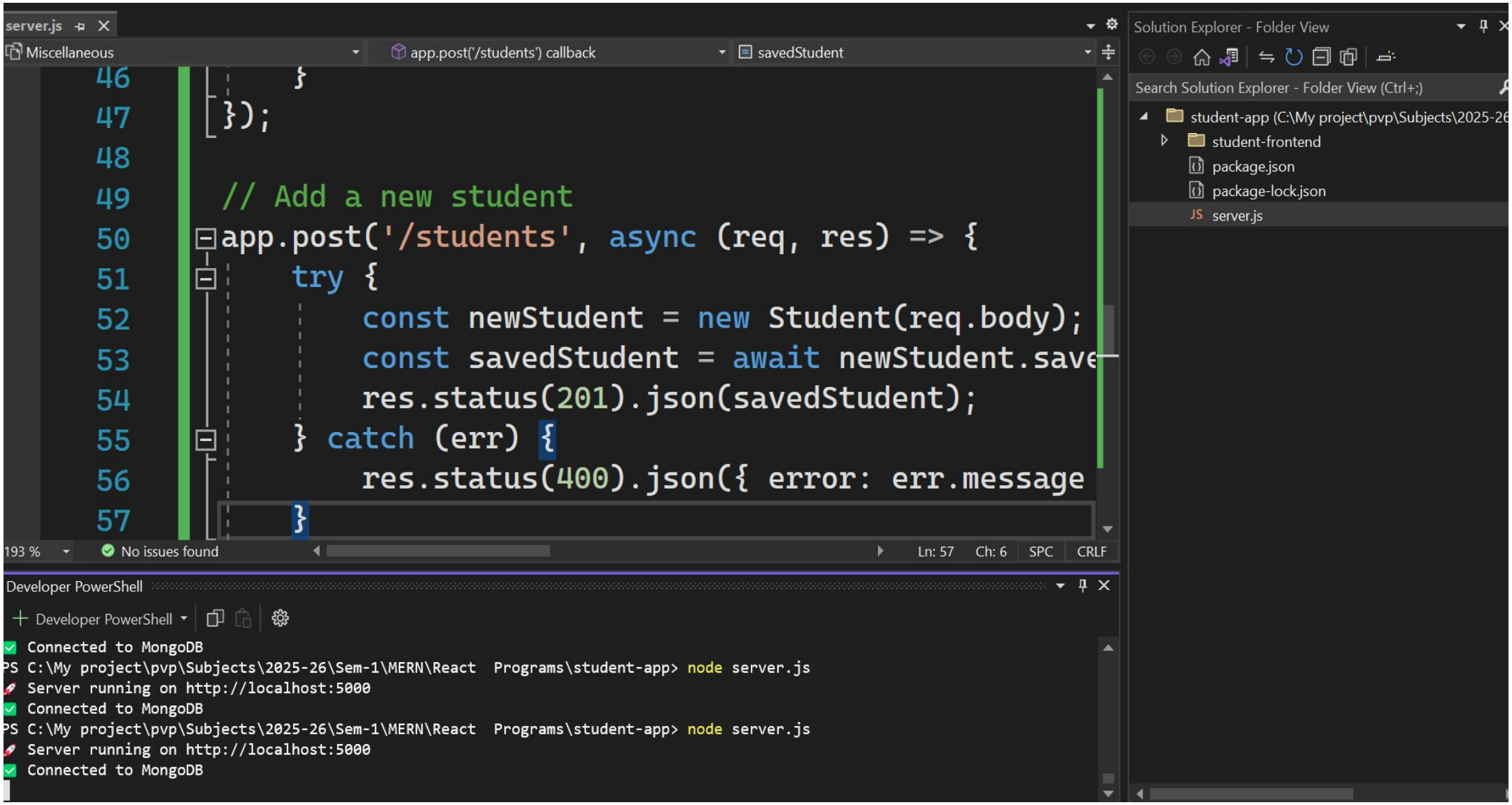
- **Handles GET requests from frontend.**
- **Student.find() → fetches all documents from MongoDB.**
- **res.json(students) → sends data back to React as JSON.**

route `/students` (POST):

- **Lets React frontend add a new student.**
- **`req.body` contains student data from frontend form.**

- **Starts Express server** on port 5000.
- Backend is now ready to handle requests from React.

Start Backend Server



Step 3: Setup Frontend (React)

1 Create React App

```
cd "C:\My project\pvp\Subjects\2025-26\Sem-1\MERN\React Programs\student-app"
npx create-react-app student-frontend
cd student-frontend
```

Update `App.js` to fetch backend data

App.js

```
import React, { useEffect, useState } from 'react';

function App() {
  const [students, setStudents] = useState([]);

  useEffect(() => {
    fetch('http://localhost:5000/students')
      .then(res => res.json())
      .then(data => setStudents(data))
      .catch(err => console.error(err));
  }, []);

  if (!students.length) return <p>Loading students...</p>;

  const keys = Object.keys(students[0]).filter(k => k !== '__v' && k !== '_id');
```

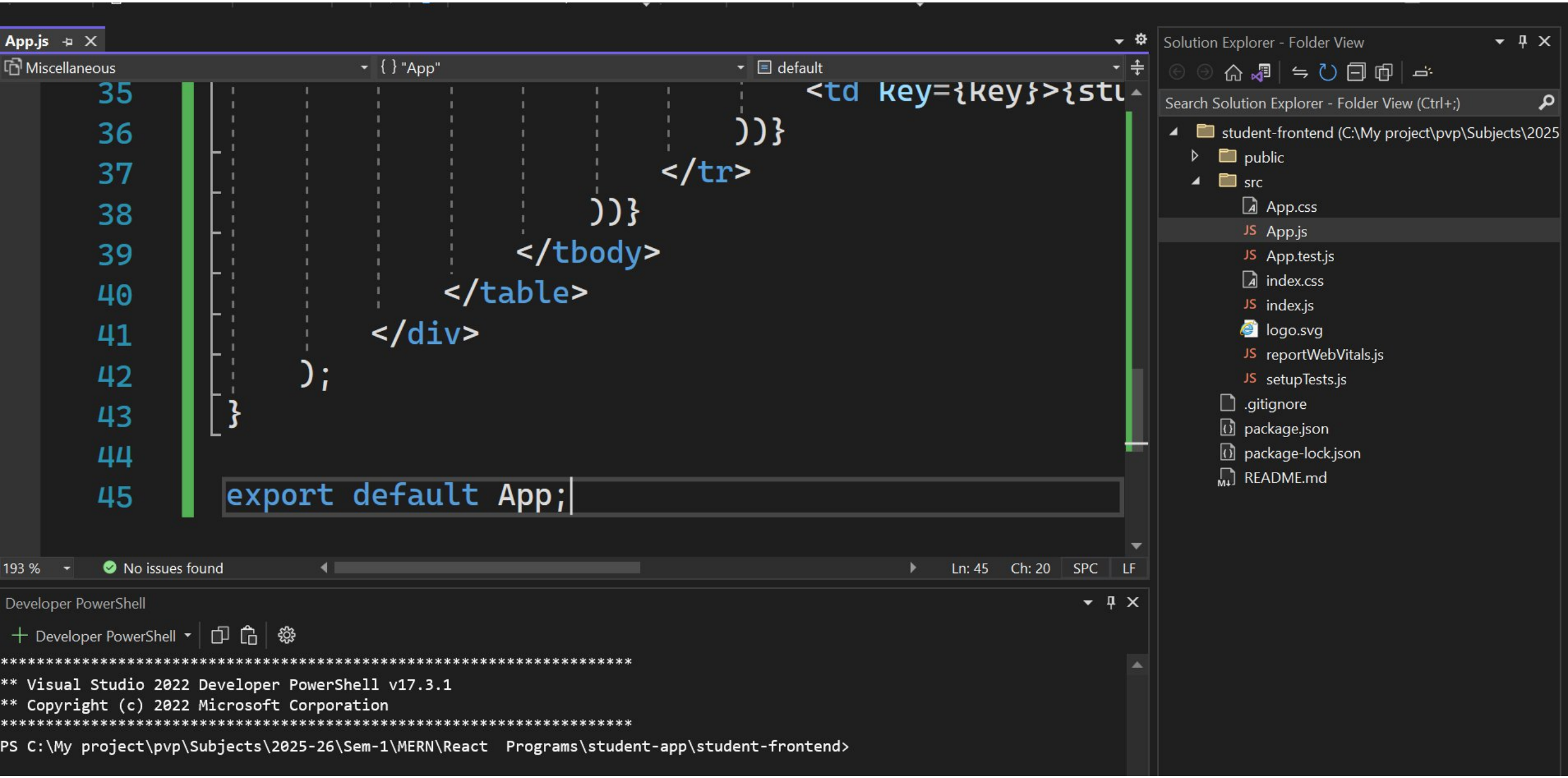
- `useState([])` → stores the list of students fetched from backend.
- `students` is the state array holding MongoDB data.

- `useEffect` → runs once when the component loads.
- `fetch('http://localhost:5000/students')` → calls backend API.
- `.then(data => setStudents(data))` → stores data in `students` state.

Dynamically generates table columns based on **keys from MongoDB documents**.

```
  return (
    <div style={{ padding: '20px' }}>
      <h1>Student Information</h1>
      <table border="1" cellPadding="10">
        <thead>
          <tr>{keys.map(key => <th key={key}>{key}</th>)}</tr>
        </thead>
        <tbody>
          {students.map(s => (
            <tr key={s._id}>
              {keys.map(key => <td key={key}>{s[key]}</td>)}
            </tr>
          ))}
        </tbody>
      </table>
    </div>
  );}export default App;
```

- **Generates HTML table dynamically from students data.**
- **keys.map → table headers.**
- **students.map → table rows.**
- **Each row uses s._id as a key (MongoDB unique ID).**


App.js
Miscellaneous
{ } "App"
default
35 <td key={key}>{stu
36))}
37 </tr>
38))}
39 </tbody>
40 </table>
41
42);
43 }
44
45 export default App;
193 %
No issues found
Ln: 45
Ch: 20
SPC
LF
Developer PowerShell
Developer PowerShell
********** Visual Studio 2022 Developer PowerShell v17.3.1
** Copyright (c) 2022 Microsoft Corporation
PS C:\My project\pvp\Subjects\2025-26\Sem-1\MERN\React Programs\student-app\student-frontend>
Solution Explorer - Folder View
Search Solution Explorer - Folder View (Ctrl+;)
student-frontend (C:\My project\pvp\Subjects\2025
public
src
App.css
App.js
App.test.js
index.css
index.js
logo.svg
reportWebVitals.js
setupTests.js
.gitignore
package.json
package-lock.json
README.md

Start React frontend

npm start

```
Compiled successfully!

You can now view student-frontend in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://10.1.53.123:3000

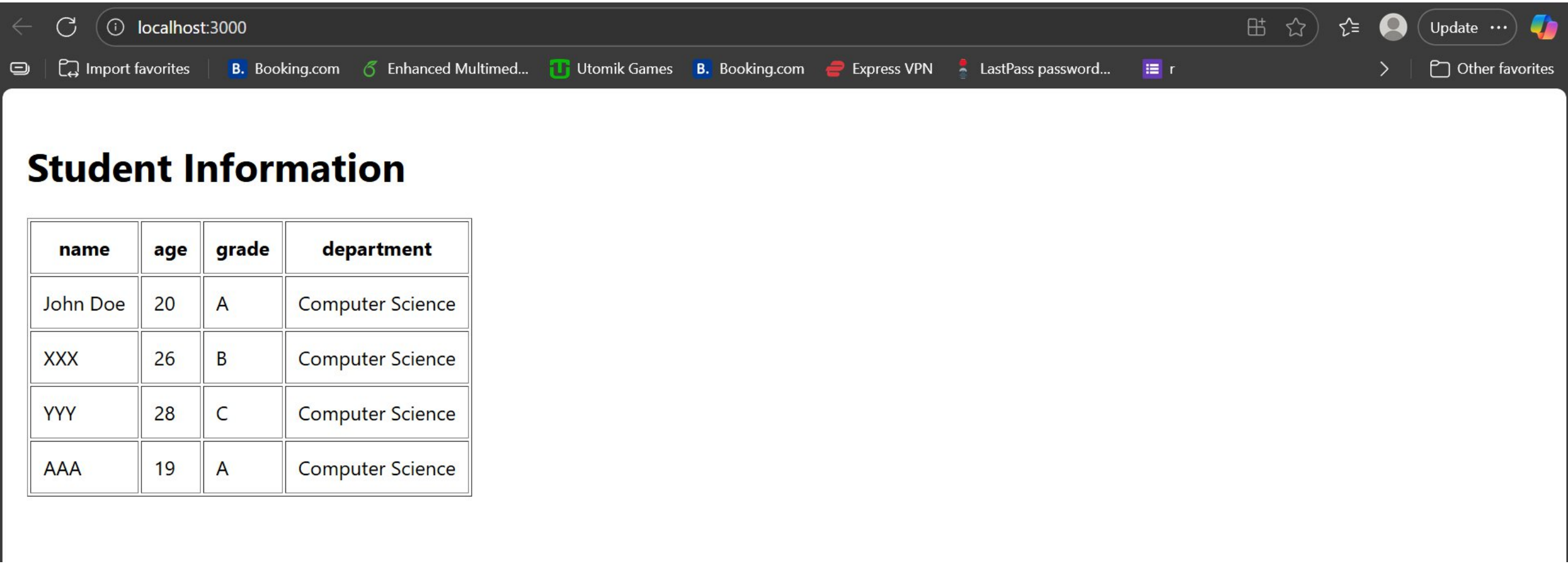
Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
Terminate batch job (Y/N)?
Terminate batch job (Y/N)? y

C:\My project\pvp\Subjects\2025-26\Sem-1\MERN\React  Programs\student-app\student-frontend>npm start

> student-frontend@0.1.0 start
> react-scripts start
```

- Opens browser: **http://localhost:3000**
- Should display **students table** fetched from your backend.

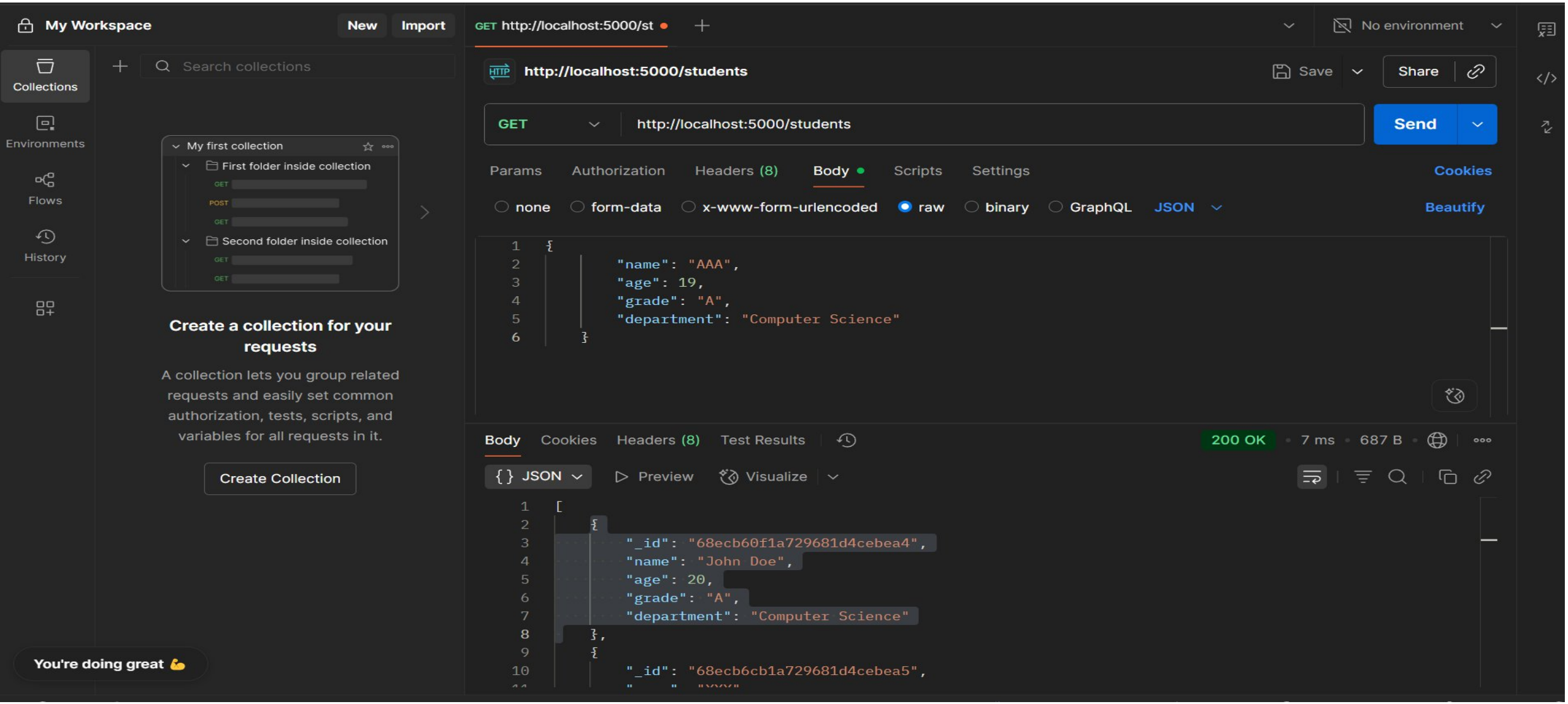


Student Information

name	age	grade	department
John Doe	20	A	Computer Science
XXX	26	B	Computer Science
YYY	28	C	Computer Science
AAA	19	A	Computer Science

Step 5 (Optional Enhancements)

- Add **Add Student** form (POST request to backend)
- Add **Delete button** (DELETE request)



```
node --version
```

```
npm --version
```

```
mkdir node-mongoDB && cd node-mongoDB
```

```
npm init -y
```

```
npm install mongodb
```

MongoDB Node.js driver:

npm install nodemon --save-dev.

To connect Node.js to a local MongoDB instance, you need to:

1. install MongoDB locally,
2. start the MongoDB server, then use the "mongodb://localhost:<port>" connection string in your Node.js code to establish a connection, where `<port>` is the port number configured for your MongoDB instance;
3. typically, you'll also need to install the MongoDB Node.js driver and
4. use the `MongoClient` class to interact with the database.