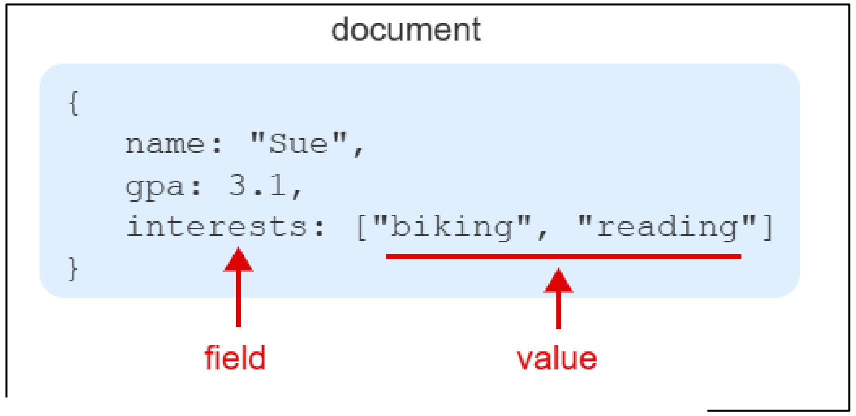
MONGO DB

MongoDB document database

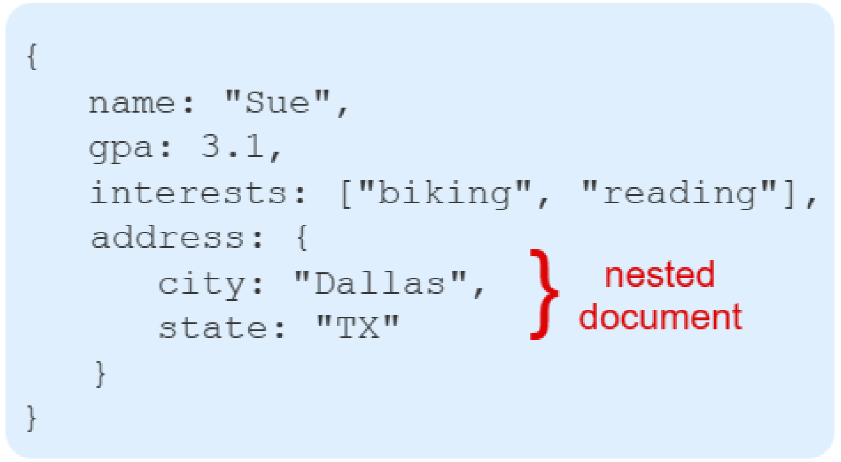
- Node.js web applications may use relational or non-relational (NoSQL) databases to store web application data. ***MongoDB*** is the most popular NoSQL database used by Node.js developers. MongoDB stores data objects as documents inside a collection.
- A ***document*** is a single data object in a MongoDB database that is composed of field/value pairs, similar to JSON property/value pairs.
- A ***collection*** is a group of related documents in a MongoDB database.
- MongoDB stores documents internally as BSON documents. A ***BSON document*** (Binary JSON) is a binary representation of JSON with additional type information. BSON types include string, integer, double, date, boolean, null, and others. A BSON document may not exceed 16 MB in size.

A single student is represented as a document with field:value pairs. The name field is assigned a BSON string, gpa is a double, and interests is an array.



```
{
   name: "Sue",
   gpa: 3.1,
   interests: ["biking", "reading"]
}
```

Documents may be nested. The student document contains a nested address document.



```
{
   name: "Sue",
   gpa: 3.1,
   interests: ["biking", "reading"],
   address: {
      city: "Dallas",
      state: "TX"
   }
}
```

collection

```
{
   name: "Sue",
   gpa: 3.1,
   interests: ["biking", "reading"]
}
```

```
{
   name: "Larry",
   gpa: 2.5,
   interests: ["RPGs", "chess"]
}
```

```
{
   name: "Anne",
   gpa: 4.0,
   interests: ["coding", "Pilates"]
}
```

MongoDB organizes documents into collections. A group of students is stored in a single collection.

Installing MongoDB

- *MongoDB runs on a wide range of operating systems. Instructions for installing MongoDB Community Edition are provided on the MongoDB website.*
- *MongoDB Shell is a program for interacting with MongoDB. Instructions for installing MongoDB Shell are also available on the MongoDB website.*

Create Database

```
Use databaseName
```

```
>use Student
```

Create Collection

```
db.createCollection(‘collectionName’)
```

```
>db.createCollection(‘students’)
```
➔ Create table

Drop Database

```
db.dropDatabase
```

```
>db.dropDatabase (‘students’)
```
➔ drop table

Inserting documents

- The insertOne() collection method inserts a single document into a collection.
- The insertMany() collection method inserts multiple documents into a collection.
- In the figure below, Sue is inserted into the students collection, then three students in the students array are inserted.

```
mydb> db.students.insertOne({ name: "Sue", gpa: 3.1 })
{
  acknowledged: true,
  insertedId: ObjectId("62794229fc4ebd4933877a9d")
}

mydb> students = [
... { name: "Maria", gpa: 4.0 },
... { name: "Xiu", gpa: 3.8 },
... { name: "Braden", gpa: 2.5 } ]

mydb> db.students.insertMany(students)
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId("627942f6fc4ebd4933877a9e"),
    '1': ObjectId("627942f6fc4ebd4933877a9f"),
    '2': ObjectId("627942f6fc4ebd4933877aa0")
  }
}
```

```
mydb> db.students.find()
[
  { _id: ObjectId("62794229fc4ebd4933877a9d"), name:
'Sue', gpa: 3.1 },
  { _id: ObjectId("627942f6fc4ebd4933877a9e"), name:
'Maria', gpa: 4 },
  { _id: ObjectId("627942f6fc4ebd4933877a9f"), name:
'Xiu', gpa: 3.8 },
  { _id: ObjectId("627942f6fc4ebd4933877aa0"), name:
'Braden', gpa: 2.5 }
]
```

The ***_id*** field is automatically assigned to every document and is always the first field in the document. The _id acts as a primary key. A ***primary key*** is a field that uniquely identifies each document in a collection. The _id may be assigned a unique value like a student ID number or use an auto-incrementing value. In the figure above, no _id field was assigned, so MongoDB automatically assigned an ObjectId to _id. An ***ObjectId*** is a 12-byte BSON type that contains a unique value. An ObjectId is displayed as a hexadecimal number. Ex: 62794229fc4ebd4933877a9d.

Finding documents

The find() collection method returns all documents by default or documents that match an optional query parameter. The findOne() collection method returns only the first document matching the query. Both methods return null if the query matches no documents.

Figure 10.9.2: Find 'Sue' and students with GPA ≥ 3.0.

```
mydb> db.students.find({ name: 'Sue' })
[
 { _id: ObjectId("62794229fc4ebd4933877a9d"), name:
'Sue', gpa: 3.1 }
]

mydb> db.students.find({ gpa: { $gte: 3.0 } })
[
 { _id: ObjectId("62794229fc4ebd4933877a9d"), name:
'Sue', gpa: 3.1 },
 { _id: ObjectId("627942f6fc4ebd4933877a9e"), name:
'Maria', gpa: 4 },
 { _id: ObjectId("627942f6fc4ebd4933877a9f"), name: 'Xiu',
gpa: 3.8 }
]
```

Operator	Description	Example

`field:value`	Matches documents with fields that are equal to the given value.	`// Matches student with this _id`
		`{ "_id" :`
		`ObjectId("62794229fc4ebd4933877a9d") }`

`$eq` `$ne`	Matches values = or ≠ to the given value.	`// Matches all docs except Sue`
		`{ name: { $ne: "Sue" } }`

`$gt` `$gte`	Matches values > or ≥ to the given value.	`// Matches students with gpa > 3.5`
		`{ gpa: { $gt: 3.5 } }`

`$lt` `$lte`	Matches values < or ≤ to the given value.	`// Matches students with gpa <= 3.0`
		`{ gpa: { $lte: 3.0 } }`

`$in` `$nin`	Matches values in or not in a given array.	`// Matches Sue, Susan, or Susie`
		`{ name: { $in: ["Sue", "Susan", "Susie"] } }`

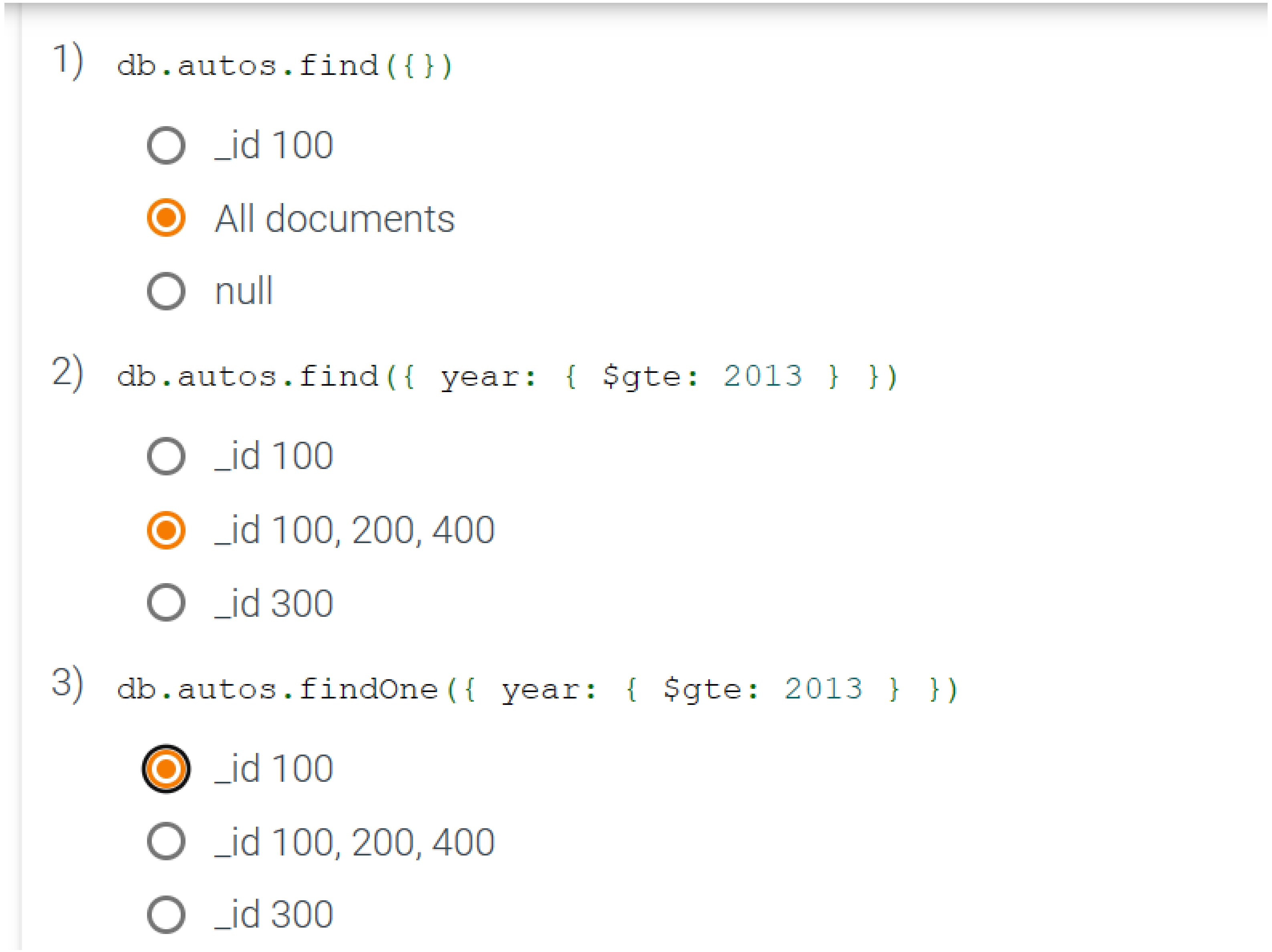
`$and`	Joins query clauses with a logical AND, returns documents that match both clauses.	`// Matches student with gpa >= 3.0 and gpa <= 3.5`
		`{ $and: [{ gpa: { $gte: 3.0 } },`
		`         { gpa: { $lte: 3.5 } }] }`

`$or`	Joins query clauses with a logical OR, returns documents that match either clauses.	`// Matches students with gpa >= 3.9 or gpa <= 3.0`
		`{ $or: [{ gpa: { $gte: 3.9 } },`
		`        { gpa: { $lte: 3.0 } }] }`

Example : Create this database and answer the one on next slide

```
[
  {
    "_id" : 100,
    "make" : "Ford",
    "model" : "Fusion",
    "year" : 2014,
    "options" : [ "engine start", "moon roof" ],
    "price" : 13500
  },
  {
    "_id" : 200,
    "make" : "Honda",
    "model" : "Accord",
    "year" : 2013,
    "options" : [ "spoiler", "alloy wheels", "sunroof" ],
    "price" : 16900
  },
```

```
{
    "_id" : 300,
    "make" : "Dodge",
    "model" : "Avenger",
    "year" : 2012,
    "options" : [ "leather seats" ],
    "price" : 10800
  },
  {
    "_id" : 400,
    "make" : "Toyota",
    "model" : "Corolla",
    "year" : 2013,
    "options" : [ "antitheft" ],
    "price" : 13400
  }
]
```



1) `db.autos.find({})`

- ○ _id 100
- [✓ All documents]
- ○ null

2) `db.autos.find({ year: { $gte: 2013 } })`

- ○ _id 100
- [✓ _id 100, 200, 400]
- ○ _id 300

3) `db.autos.findOne({ year: { $gte: 2013 } })`

- [✓ _id 100]
- ○ _id 100, 200, 400
- ○ _id 300

4)
```
db.autos.findOne({ year: { $gte: 2016 } })
```

- _id 100
- Run-time error
- [✓ null]

5)
```
db.autos.find({ $and: [ {price: { $lte: 15000 } },
   { options: { $in: ["sunroof", "antitheft", "moon
roof"]
   } } ] })
```

- _id 100
- [✓ _id 100, 400]
- _id 100, 200, 400

6)
```
db.autos.find({ $or: [ {"make":"Honda"},
   { year: {$ne: 2013} } ] })
```

- All documents
- _id 100, 300
- [✓ _id 100, 200, 300]

Updating documents

- The updateOne() collection method modifies a single document in a collection. The updateMany() collection method modifies multiple documents in a collection. The methods have two required parameters:
  - **query** - The query to find the document(s) to update. An empty query {} matches all documents.
  - **update** - The modification to perform on matched documents using an update operator like $inc, $set, and $unset.
- In the example below, the calls to updateOne() and updateMany() return the matchedCount property indicating how many documents matched the query, and the modifiedCount property indicating the number of documents modified.

Change Sue's GPA to 3.3, and set all students with GPA > 3 to 1.

```
mydb> db.students.updateOne({ name: 'Sue' }, { $set: { gpa: 3.3 } })
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}

mydb> db.students.find({ name: 'Sue' })
{ "_id" : ObjectId("5e600d18bbd10ee972f6ed9a"), "name" : "Sue",
"gpa" : 3.3 }
```

E.g. To update student table and add admission Date as current date:

```
Db.Students.updateMany({} , {$set:{admissionDate:new Date()}})
```

```
mydb> db.students.updateMany({ gpa: { $gt: 3 } }, { $set: { gpa: 1 } })
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 3,
  modifiedCount: 3,
  upsertedCount: 0
}

mydb> db.students.find()
[
  { _id: ObjectId("62794229fc4ebd4933877a9d"), name: 'Sue', gpa: 1 },
  { _id: ObjectId("627942f6fc4ebd4933877a9e"), name: 'Maria', gpa: 1 },
  { _id: ObjectId("627942f6fc4ebd4933877a9f"), name: 'Xiu', gpa: 1 },
  { _id: ObjectId("627942f6fc4ebd4933877aa0"), name: 'Braden', gpa: 2.5 }
]
```

Common MongoDB update operators

Operator	Description	Example

`$currentDate`	Sets a field's value to the current date/time	`// Sue's "birthDate" : ISODate("2022-05-09T16:39:00.121Z")`
		`db.students.updateOne({ name: 'Sue' },`
		`   { $currentDate: { birthDate: true } })`

`$inc`	Increments a field's value by the specified amount	`// Sue's "gpa" incremented from 3.1 to 3.2`
		`db.students.updateOne({ name: 'Sue' },`
		`   { $inc: { gpa: 0.1 } })`

`$rename`	Renames a field	`// Sue's "name" is now "firstName"`
		`db.students.updateOne({ name: 'Sue' },`
		`   { $rename: { name: 'firstName' } })`

`$set`	Sets a field's value	`// Sue's "gpa" : 4.0`
		`db.students.updateOne({ name: 'Sue' },`
		`   { $set: { gpa: 4.0 } })`

`$unset`	Removes a field	`// Removes Sue's "gpa" and "birthDate" fields`
		`db.students.updateOne({ name: 'Sue' },`
		`   { $unset: { gpa: "", birthDate: "" } })`

Refer to the "autos" collection below, and choose the result of each command.

```
[
  {
    "_id" : 100,
    "make" : "Ford",
    "model" : "Fusion",
    "year" : 2014,
    "options" : [ "engine start", "moon roof" ],
    "price" : 13500
  },
  {
    "_id" : 200,
    "make" : "Honda",
    "model" : "Accord",
    "year" : 2013,
    "options" : [ "spoiler", "alloy wheels", "sunroof" ],
    "price" : 16900
  }
]
```

1)
```
db.autos.updateOne({ price: { $gt: 10000} },
   { $set: { year: 2000, options: [] }})
```

- [✓ Only auto with _id 100 has year set to 2000 and options removed.]
- Both autos have year set to 2000 and options removed.
- No autos are updated.

Correct

updateOne() only updates the first document that has a price > 10000.

2)
```
db.autos.updateMany({ price: { $gt: 10000} },
   { $set: { year: 2000, options: [] }})
```

- Only auto with _id 100 has year set to 2000 and options removed.
- [✓ Both autos have year set to 2000 and options removed.]
- No autos are updated.

Correct

updateMany() updates all documents that have price > 1000, so both autos are updated.

3)
```
db.autos.updateOne({ price: { $gt: 10000} },
   { $set: { sold: true }})
```

- No autos are updated because the autos do not have a "sold" field.
- [✓ Auto with _id 100 has new field "sold" set to true.]
- Both autos have a new field "sold" set to true.

Correct

The $set operator creates a new sold field since the field did not previously exist.

4)
```
db.autos.updateOne({ _id: 100 },
   { $currentDate: { soldDate: true }})
```

- Auto with _id 100 has new field "soldDate" set to true.
- Auto with _id 100 has new field "soldDate" set to the Unix epoch (January 1, 1970).
- [✓ Auto with _id 100 has new field "soldDate" set to the current date and time.]

Correct

The soldDate appears in ISO 8601 datetime format. Ex: ISODate("2022-05-10T23:21:29.439Z").

5)
```
db.autos.updateOne({ _id: 100 },
   { $inc: { price: -500, year: 2 }})
```

- [✓ Auto with _id 100 has price reduced by 500 and year increased by 2.]
- Auto with _id 100 has price set to -500 and year set to 2.
- Auto with _id 100 has price and year fields removed.

Correct

The $inc operator increments price by -500, setting the price to 13500 + -500 = 13000. The year is incremented by 2, setting year to 2014 + 2 = 2016.

Deleting documents

- The deleteOne() collection method deletes a single document from a collection. The deleteMany() collection method deletes multiple document from a collection. The methods have a required query parameter that matches documents to delete.

- In the example below, the calls to deleteOne() and deleteMany() return a deletedCount property indicating how many documents were deleted.

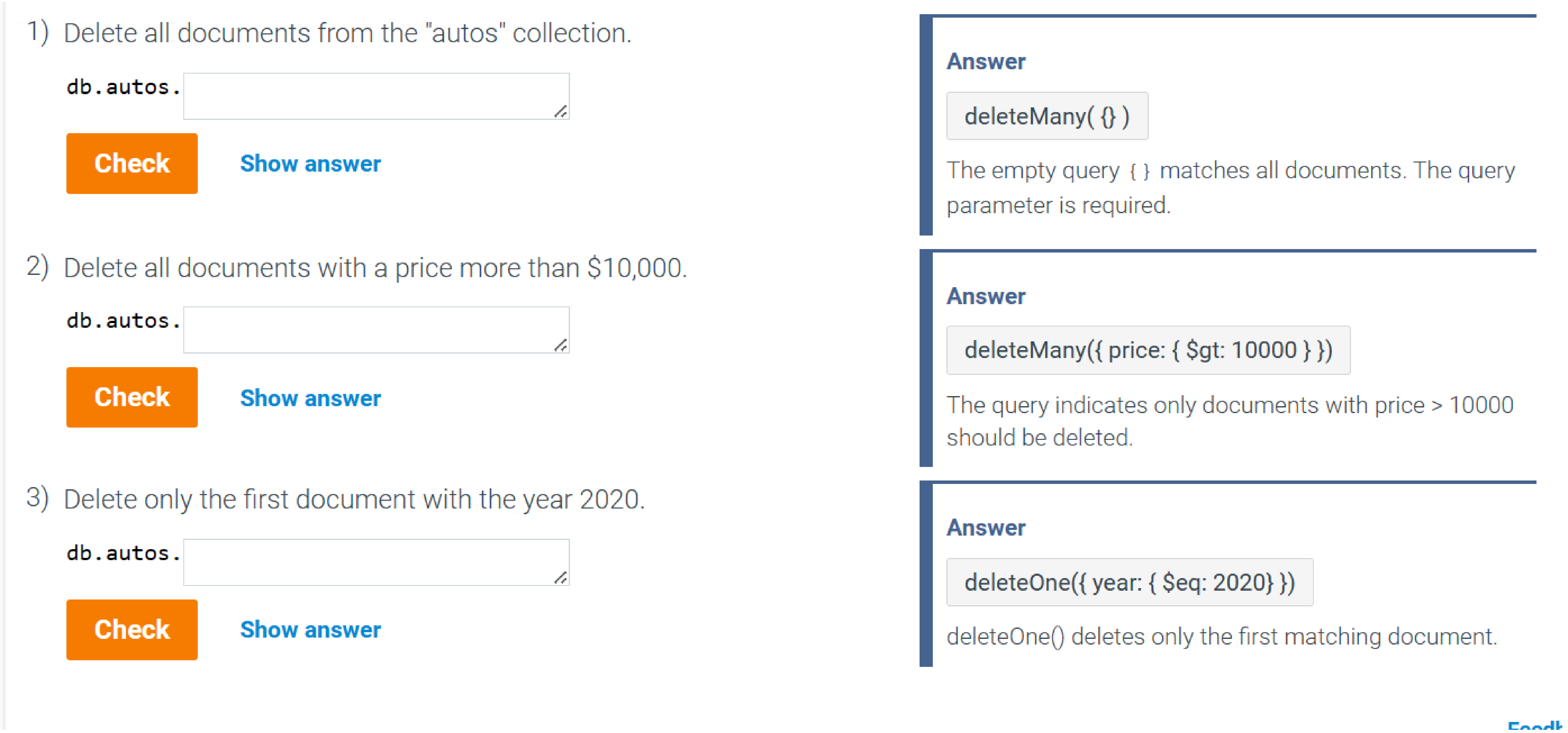
Delete the first student with GPA < 3.5 (Sue), and delete all students with GPA > 3.5 (Maria and Xiu).

```
mydb> db.students.deleteOne({ gpa: { $lt:3.5 } })
{ acknowledged: true, deletedCount: 1 }

mydb> db.students.find()
[
 { _id: ObjectId("627942f6fc4ebd4933877a9e"), name:
'Maria', gpa: 4 },
 { _id: ObjectId("627942f6fc4ebd4933877a9f"), name: 'Xiu',
gpa: 3.8 },
 { _id: ObjectId("627942f6fc4ebd4933877aa0"), name:
'Braden', gpa: 2.5 }
]

mydb> db.students.deleteMany({ gpa: { $gt:3.5 } })
{ acknowledged: true, deletedCount: 2 }

mydb> db.students.find()
[
 { _id: ObjectId("627942f6fc4ebd4933877aa0"), name:
'Braden', gpa: 2.5 }
]
```



1) Delete all documents from the "autos" collection.

```
db.autos.
```

Check   Show answer

Answer

```
deleteMany( {} )
```

The empty query `{ }` matches all documents. The query parameter is required.

2) Delete all documents with a price more than $10,000.

```
db.autos.
```

Check   Show answer

Answer

```
deleteMany({ price: { $gt: 10000 } })
```

The query indicates only documents with price > 10000 should be deleted.

3) Delete only the first document with the year 2020.

```
db.autos.
```

Check   Show answer

Answer

```
deleteOne({ year: { $eq: 2020} })
```

deleteOne() deletes only the first matching document.

Feedb