Express.js with MongoDB interview questions, along with concise and clear **answers**.

These cover beginner to intermediate topics and are useful for both developers and interviewers.

□ Basic Level

1. What is Express.js?

Answer:
Express.js is a minimal and flexible Node.js web application framework that provides features for building web and mobile applications, such as routing, middleware support, and HTTP utilities.

---

2. How do you connect MongoDB with Express.js?

Answer:
Using the `mongoose` library:

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

mongoose.connect('mongodb://localhost:27017/mydb', {
  useNewUrlParser: true,
  useUnifiedTopology: true,
}).then(() => console.log('Connected to MongoDB'));
```

---

3. What is Mongoose? Why use it?

Answer:
Mongoose is an ODM (Object Data Modeling) library for MongoDB and Node.js. It provides a schema-based solution to model your application data with built-in validation, type casting, and query building.

---

4. What is middleware in Express.js?

Answer:
Middleware are functions that execute during the request-response cycle. They can:

- Modify `req` or `res` objects
- End the request-response cycle
- Call the next middleware

Example:

```
app.use((req, res, next) => {
  console.log('Middleware hit');
  next();
});
```

---

5. How do you define a route in Express?

Answer:

```
app.get('/api/users', (req, res) => {
  res.send('Users route');
});
```

□ Intermediate Level

6. How do you define a Mongoose schema and model?

Answer:

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

const userSchema = new mongoose.Schema({
  name: String,
  email: { type: String, unique: true },
});

const User = mongoose.model('User', userSchema);
```

---

7. How can you handle errors globally in Express?

Answer:
By using an error-handling middleware:

```
app.use((err, req, res, next) => {
```

```
  console.error(err.stack);
  res.status(500).json({ message: 'Internal Server Error' });
});
```

---

8. How do you perform CRUD operations using Express and MongoDB?

Answer:

- **Create**:

```
app.post('/users', async (req, res) => {
  const user = new User(req.body);
  await user.save();
  res.status(201).json(user);
});
```

- **Read**:

```
app.get('/users', async (req, res) => {
  const users = await User.find();
  res.json(users);
});
```

- **Update**:

```
app.put('/users/:id', async (req, res) => {
  const user = await User.findByIdAndUpdate(req.params.id, req.body, { new:
true });
  res.json(user);
});
```

- **Delete**:

```
app.delete('/users/:id', async (req, res) => {
  await User.findByIdAndDelete(req.params.id);
  res.json({ message: 'User deleted' });
});
```

---

9. What is the difference between `res.send()`, `res.json()`, and `res.status()`?

Answer:

- `res.send()` can send any type (string, buffer, etc.).

- `res.json()` sends a JSON response.
- `res.status()` sets the HTTP status code.

Example:

```
res.status(200).json({ message: 'OK' });
```

---

10. How do you protect routes using middleware (e.g., JWT auth)?

Answer:

```
const auth = (req, res, next) => {
  const token = req.headers.authorization?.split(' ')[1];
  if (!token) return res.status(401).json({ message: 'Unauthorized' });
  try {
    const user = jwt.verify(token, 'secret');
    req.user = user;
    next();
  } catch (err) {
    res.status(403).json({ message: 'Forbidden' });
  }
};

app.get('/protected', auth, (req, res) => {
  res.send('Protected data');
});
```

□ Advanced Level

11. What is population in Mongoose?

Answer:
Population is a way to automatically replace a referenced document's `_id` with the actual document.

Example:

```
User.find().populate('profile').exec();
```

---

12. What are the different ways to query MongoDB using Mongoose?

Answer:

- `Model.find()`
- `Model.findById()`
- `Model.findOne()`
- Query chaining: `User.find().where('age').gt(18).sort('name')`

---

13. What is the use of `.lean()` in Mongoose?

Answer:
`.lean()` returns plain JavaScript objects instead of Mongoose documents. It improves performance and reduces memory usage for read-heavy operations.

---

14. How do you handle validation in Mongoose?

Answer:
Use schema-level validation:

```
const userSchema = new mongoose.Schema({
  email: { type: String, required: true, match: /.+\@.+\..+/ },
});
```

You can also catch validation errors in try-catch:

```
try {
  await user.save();
} catch (err) {
  if (err.name === 'ValidationError') {
    console.log(err.message);
  }
}
```

---

15. What are some best practices when using Express and MongoDB together?

Answer:

- Use environment variables (`dotenv`) for secrets.
- Separate routes, models, and controllers.

- Use async/await and error handling middleware.
- Use indexes in MongoDB for faster querying.
- Validate inputs (e.g., with `Joi` or `express-validator`).
- Sanitize user input to prevent NoSQL injection.

In **Mongoose**, there are several powerful and flexible ways to **query MongoDB**. These allow you to retrieve documents based on various criteria and use advanced features like filtering, sorting, pagination, population, and more.

□ 1. Basic Query Methods

Method	Description
`find()`	Returns an array of documents that match a filter.
`findOne()`	Returns the first document that matches the filter.
`findById()`	Returns a document by its `_id`.
`countDocuments()`	Returns the number of documents that match a filter.
`distinct()`	Returns an array of distinct values for a field.

□ Examples:

```
User.find({ age: { $gt: 18 } });           // All users with age > 18
User.findOne({ email: 'test@example.com' }); // One user with this email
User.findById('6520c9fbb362cf25db3e6b90');   // Find by _id
User.countDocuments({ isActive: true });     // Count active users
User.distinct('country');                    // Unique list of countries
```

□ 2. Query with Comparison Operators

Operator	Description
`$eq`	Equal to
`$ne`	Not equal to
`$gt`	Greater than
`$gte`	Greater than or equal to
`$lt`	Less than
`$lte`	Less than or equal to
`$in`	In array
`$nin`	Not in array

□ Example:

```
User.find({ age: { $gte: 18, $lte: 30 } });
User.find({ status: { $in: ['active', 'pending'] } });
```

□ 3. Query Chaining

You can chain multiple query helpers for more control:

```
User.find({ isActive: true })
    .where('age').gt(18)
    .sort('name')
    .limit(10)
    .skip(20); // For pagination
```

□ 4. Logical Operators

Operator	Description
`$or`	At least one matches
`$and`	All must match
`$nor`	None must match
`$not`	Inverts query logic

□ Example:

```
User.find({
  $or: [
    { role: 'admin' },
    { role: 'moderator' }
  ]
});
```

□ 5. Regex and Pattern Matching

```
User.find({ name: { $regex: /^John/i } }); // Names starting with "John" (case-insensitive)
```

□ 6. Populate (for referenced documents)

If you use MongoDB references (`ref`), you can populate them:

```
Post.find().populate('author'); // Replace author _id with actual document
```

□ 7. Lean Query

Return plain JS objects instead of full Mongoose documents (faster read):

```
User.find().lean();
```

□ 8. Aggregation Framework

For complex queries, transformations, and grouping:

```
User.aggregate([
  { $match: { age: { $gte: 18 } } },
  { $group: { _id: "$country", count: { $sum: 1 } } },
]);
```

Summary Table:

Query Type	Method Example
Find Many	`User.find({ age: { $gt: 18 } })`
Find One	`User.findOne({ email: 'a@b.com' })`
Find by ID	`User.findById(id)`
Filter & Sort	`User.find().where('age').gt(18).sort()`
Pagination	`User.find().limit(10).skip(20)`
Regex	`User.find({ name: /john/i })`
Count	`User.countDocuments({ active: true })`
Populate	`Post.find().populate('user')`
Aggregate	`User.aggregate([...])`

---

very commonly used Mongoose methods:

□ **`findByIdAndUpdate()`**

□ **`findByIdAndDelete()`**

These are **built-in Mongoose methods** that allow you to update or delete documents directly by their `_id`.

□ `findByIdAndUpdate()`

Syntax:

```
Model.findByIdAndUpdate(id, update, options)
```

- **`id`**: The document's `_id`
- **`update`**: An object with fields to update
- **`options`** *(optional)*: Commonly used option is `{ new: true }` to return the updated document

□ Example:

```
const updatedUser = await User.findByIdAndUpdate(
  '6525b1ec5d3f9c16a4d78900',
  { age: 30 },
  { new: true } // returns updated document
);

console.log(updatedUser);
```

□ `findByIdAndDelete()`

Syntax:

```
Model.findByIdAndDelete(id)
```

- Deletes the document by `_id`
- Returns the deleted document

□ Example:

```
const deletedUser = await User.findByIdAndDelete('6525b1ec5d3f9c16a4d78900');

console.log(deletedUser); // Deleted document
```

□ Important Notes:

- These methods use **MongoDB's `findOneAndUpdate` and `findOneAndDelete`** behind the scenes.
- You should handle cases when the document isn't found (i.e., returns `null`).
- If you need **validation** during update, use:

```
{ new: true, runValidators: true }
```

□ Alternative Query Names

Task	Mongoose Method	Aliases/Notes
Update by ID	`findByIdAndUpdate()`	No alias
Delete by ID	`findByIdAndDelete()`	Alias: `findByIdAndRemove()`

□ `findByIdAndRemove()` is deprecated in practice — use `findByIdAndDelete()`.