**MongoDB Introduction**

MongoDB is a popular open-source, NoSQL database that provides high performance, high availability, and easy scalability. It stores data in flexible, JSON-like documents called BSON (Binary JSON), making it suitable for a wide range of use cases, including real-time analytics, content management, mobile applications, and more. MongoDB uses a flexible schema, allowing you to store heterogeneous data in the same collection without requiring a predefined schema.

Here's an introduction to MongoDB's key concepts and features:

1. Document-Oriented: MongoDB is a document-oriented database, meaning it stores data in collections of JSON-like documents. Each document can have a different structure, and fields can vary between documents in the same collection. This flexibility allows you to store data in a way that closely mirrors your application's data model.
2. Collections and Documents: In MongoDB, data is organised into collections, which are analogous to tables in relational databases. Each collection contains multiple documents, which are individual records or data entries. Documents are composed of field-value pairs and support rich data structures like arrays and nested documents.
3. Schemaless Design: MongoDB's schema less design allows for dynamic and flexible data modelling. Unlike relational databases, where schemas are predefined and enforced, MongoDB allows you to store data without a fixed schema. This flexibility simplifies development and accommodates evolving data models.
4. JSON/BSON Format: MongoDB stores data in BSON (Binary JSON), a binary-encoded serialisation of JSON-like documents. BSON extends the JSON model with additional data types and optimizations for storage and indexing. BSON documents are efficient to serialise and deserialize, making MongoDB suitable for high-performance applications.
5. Indexing and Querying: MongoDB supports indexing to optimise query performance. Indexes can be created on single fields, compound fields, and arrays. MongoDB provides a powerful query language with support for a wide range of operations, including filtering, sorting, aggregation, and geospatial queries.
6. Replication and High Availability: MongoDB supports replica sets, which are groups of MongoDB instances that maintain copies of the same data. Replica sets provide redundancy and fault tolerance, allowing MongoDB to continue operating even in the event of hardware failures or network partitions. MongoDB automatically elects a primary node to serve read and write operations, with secondary nodes replicating data asynchronously.
7. Scalability: MongoDB scales horizontally through sharding, a technique for distributing data across multiple servers. Sharding allows MongoDB to handle large volumes of data and high throughput by distributing data and query load across multiple nodes in a cluster.
8. Aggregation Framework: MongoDB includes a powerful aggregation framework for performing complex data processing and analysis tasks. The aggregation framework supports pipeline-based data transformation, enabling operations like filtering, grouping, sorting, and calculating aggregations.

Overall, MongoDB is a versatile and feature-rich database that offers flexibility, scalability, and performance for modern application development. Its document-oriented design, flexible schema, and rich query capabilities make it well-suited for a wide range of use cases in both small-scale and enterprise applications.

**Setup MongoDB**

To set up MongoDB on your system, you'll need to follow these general steps:

* **Download MongoDB:** Visit the official MongoDB website (https://www.mongodb.com/try/download/community) and download the appropriate MongoDB Community Server for your operating system (Windows, macOS, or Linux).
* **Install MongoDB:** Follow the installation instructions provided for your operating system. These typically involve running an installer or extracting the downloaded files to a directory on your system.
* **Start MongoDB Server:** After installation, you'll need to start the MongoDB server. The exact steps depend on your operating system:
  + Windows: Open a Command Prompt window and navigate to the directory where MongoDB is installed. Run the mongod command to start the MongoDB server.
  + macOS and Linux: Open a terminal window and run the mongod command to start the MongoDB server.

By default, MongoDB will store data in the /data/db directory. If this directory doesn't exist, you may need to create it manually.

* **Connect to MongoDB:** Once the MongoDB server is running, you can connect to it using the MongoDB shell or a MongoDB client application. To connect using the MongoDB shell, open a new terminal window and run the mongo command. This will start the MongoDB shell and connect to the local MongoDB server. Alternatively, you can use a graphical user interface (GUI) tool like MongoDB Compass to connect to and interact with MongoDB databases.
* **Create Databases and Collections:** With MongoDB connected, you can create databases and collections as needed. You can create a new database using the use command and switch to it. For example:

| use mydatabase |
| --- |

You can then create collections within the database and perform CRUD (Create, Read, Update, Delete) operations on them using MongoDB commands or client applications.

* Configure MongoDB (Optional): Depending on your requirements, you may need to configure MongoDB settings such as authentication, replication, sharding, and security. Refer to the MongoDB documentation for detailed instructions on configuring MongoDB for your specific use case.

By following these steps, you can set up MongoDB on your system and start using it to store and manage your data. Additionally, MongoDB offers cloud-hosted solutions like MongoDB Atlas, which provide managed MongoDB instances with additional features such as automated backups, monitoring, and scaling.

**MongoDB Atlas**

MongoDB Atlas is a fully-managed, cloud-hosted database service provided by MongoDB, Inc. It allows users to deploy, manage, and scale MongoDB databases with ease, without the need to handle infrastructure management tasks such as server provisioning, configuration, and maintenance. MongoDB Atlas offers a range of features and capabilities designed to streamline database operations and provide a robust and reliable database platform for modern applications.

1. **Automated Provisioning and Scaling:** MongoDB Atlas automates the process of provisioning and scaling MongoDB clusters, allowing users to deploy new clusters with just a few clicks. It supports flexible scaling options, including vertical scaling (resizing individual cluster instances) and horizontal scaling (adding or removing cluster nodes).
2. **High Availability and Fault Tolerance:** MongoDB Atlas provides built-in high availability and fault tolerance features to ensure continuous uptime and data durability. It uses replica sets to replicate data across multiple nodes within a cluster, allowing for automatic failover and data redundancy in the event of hardware failures or network partitions.
3. **Security and Compliance:** MongoDB Atlas includes comprehensive security features to protect data and ensure compliance with regulatory requirements. It supports encryption at rest and in transit, role-based access control (RBAC), network isolation using Virtual Private Clouds (VPCs), and auditing for monitoring database activity.
4. **Managed Backup and Restore:** MongoDB Atlas offers managed backup and restore functionality to simplify data protection and disaster recovery. It allows users to schedule automated backups, retain backups for extended periods, and perform point-in-time restores to recover data to a specific point in time.
5. **Monitoring and Performance Optimization:** MongoDB Atlas provides monitoring and performance optimization tools to help users monitor database performance, identify bottlenecks, and optimize query performance. It includes built-in performance metrics, query profiling, and index usage statistics to help users optimize their MongoDB deployments for performance and scalability.
6. **Global Deployment and Multi-Cloud Support:** MongoDB Atlas supports global deployment across multiple regions and cloud providers, allowing users to deploy clusters closer to their users for lower latency and improved performance. It offers support for leading cloud platforms such as AWS, Azure, and Google Cloud Platform (GCP), as well as multi-cloud clusters for increased resilience and flexibility.

# **Connect MongoDB With NodeJS Application**

To connect MongoDB with the MongoDB driver in a Node.js application, you can use the official MongoDB Node.js driver, which provides a native JavaScript interface for interacting with MongoDB databases. Below is a basic example of how to connect to a MongoDB database using the MongoDB driver in Node.js

1. **Install the MongoDB Node.js Driver:** First, install the MongoDB Node.js driver using npm (Node.js Package Manager) by running the following command in your project directory:

| npm install mongodb |
| --- |

1. **Create a Connection to MongoDB:** In your Node.js application, create a connection to your MongoDB database using the MongoClient class provided by the MongoDB driver

| const { MongoClient } = require('mongodb');  // Connection URI const uri = 'mongodb://localhost:27017/mydatabase';  // Create a new MongoClient const client = new MongoClient(uri, { useNewUrlParser: true, useUnifiedTopology: true });  // Connect to MongoDB async function connectToMongoDB() {  try {  // Connect to MongoDB server  await client.connect();  console.log('Connected to MongoDB');   // Access the database and perform operations  const db = client.db();  // Perform database operations here   } catch (error) {  console.error('Error connecting to MongoDB:', error);  } }  // Call the connectToMongoDB function to establish the connection connectToMongoDB();  const { MongoClient } = require('mongodb');  // Connection URI const uri = 'mongodb://localhost:27017/mydatabase';  // Create a new MongoClient const client = new MongoClient(uri, { useNewUrlParser: true, useUnifiedTopology: true });  // Connect to MongoDB async function connectToMongoDB() {  try {  // Connect to MongoDB server  await client.connect();  console.log('Connected to MongoDB');   // Access the database and perform operations  const db = client.db();  // Perform database operations here   } catch (error) {  console.error('Error connecting to MongoDB:', error);  } }  // Call the connectToMongoDB function to establish the connection connectToMongoDB(); |
| --- |

3.**Perform Database Operations:** Once the connection is established, you can access the database using the db() method on the MongoClient object (client). From there, you can perform various database operations such as inserting documents, querying data, updating documents, and deleting documents using the MongoDB Node.js driver's API.

4. **Close the Connection (Optional):** After completing your database operations, remember to close the MongoDB connection to free up resources:

| // Close the MongoDB connection async function closeMongoDBConnection() {  try {  await client.close();  console.log('Disconnected from MongoDB');  } catch (error) {  console.error('Error closing MongoDB connection:', error);  } }  // Call the closeMongoDBConnection function to close the connection closeMongoDBConnection(); |
| --- |

**Sample Queries**

MongoDB offers a rich set of query capabilities to interact with your data. Below are some important queries commonly used in MongoDB:

1. **Querying Documents:** MongoDB provides the find() method to query documents from a collection. You can specify criteria to filter documents based on specific field values.

| // Find all documents in a collection db.collection('users').find();  // Find documents that match a specific criteria db.collection('users').find({ age: { $gt: 25 } }); // Find users older than 25 |
| --- |

1. **Counting Documents:** You can use the count() method to count the number of documents in a collection that match a query.

| // Count the number of documents in a collection db.collection('users').count();  // Count the number of documents that match a specific criteria db.collection('users').count({ age: { $gt: 25 } }); // Count users older than 25 |
| --- |

1. **Sorting Documents:** MongoDB allows you to sort query results based on one or more fields. You can use the sort() method to specify sorting criteria.

| // Sort documents in ascending order by a field db.collection('users').find().sort({ name: 1 }); // Sort by name in ascending order  // Sort documents in descending order by a field db.collection('users').find().sort({ age: -1 }); // Sort by age in descending order |
| --- |

1. **Limiting Results:** You can limit the number of documents returned by a query using the limit() method.

| // Limit the number of documents returned by a query db.collection('users').find().limit(10); // Return only the first 10 documents |
| --- |

1. **Skipping Results:** MongoDB allows you to skip a specified number of documents from the beginning of the result set using the skip() method.

| // Skip a specified number of documents from the beginning of the result set db.collection('users').find().skip(10); // Skip the first 10 documents |
| --- |

1. **Aggregation Framework:** MongoDB provides a powerful aggregation framework for performing complex data processing tasks like grouping, sorting, filtering, and computing aggregations. You can use the aggregate() method to perform aggregation operations.

| // Perform aggregation operations db.collection('orders').aggregate([  { $group: { \_id: '$product', totalAmount: { $sum: '$amount' } } }, // Group by product and calculate total amount  { $sort: { totalAmount: -1 } } // Sort by total amount in descending order ]); |
| --- |

1. **Indexing:** Indexes improve query performance by allowing MongoDB to quickly locate documents in a collection. You can create indexes using the createIndex() method.

| // Create an index on a field db.collection('users').createIndex({ email: 1 }); // Create an index on the 'email' field |
| --- |

# **Installation of Mongoose & Connection**

To install Mongoose and establish a connection with MongoDB using Mongoose in a Node.js application, follow these steps:

1. **Install Mongoose:** First, you need to install Mongoose package in your Node.js project. You can do this using npm (Node Package Manager) by running the following command in your terminal:

| npm install mongoose |
| --- |

1. **Require Mongoose in your Node.js application:** Once Mongoose is installed, you need to require it in your Node.js application to use it. You typically do this at the top of your JavaScript file:

| const mongoose = require('mongoose'); |
| --- |

1. **Establish Connection with MongoDB:** Next, you'll need to establish a connection with your MongoDB database using Mongoose. You can use the mongoose.connect() method to connect to your MongoDB database. Replace <connection\_string> with your actual MongoDB connection string.

| // Connect to MongoDB using Mongoose  // Replace <connection\_string> with your MongoDB connection string. You // can obtain this connection string from MongoDB Atlas or your local //MongoDB server. mongoose.connect('<connection\_string>', {  useNewUrlParser: true,  useUnifiedTopology: true,  useCreateIndex: true, // (Optional) Ensures that indexes are created for any schema indexes defined  useFindAndModify: false // (Optional) Disables the use of findOneAndUpdate() and findOneAndDelete() functions (deprecated in Mongoose 6.x) }) .then(() => {  console.log('Connected to MongoDB'); }) .catch((error) => {  console.error('Error connecting to MongoDB:', error); }); |
| --- |

# **Schema & Sample Queries Using Mongoose**

In Mongoose, models are used to define the structure and behavior of MongoDB documents. They provide a schema definition, data validation, and methods for interacting with MongoDB collections. Below are the steps to define and use models in Mongoose

1. **Define a Schema:** First, you need to define a schema that specifies the structure of your MongoDB documents. A schema defines the fields and their types, along with any validation rules or default values.

| const mongoose = require('mongoose');  // Define a schema for a user document const userSchema = new mongoose.Schema({  name: { type: String, required: true },  email: { type: String, required: true, unique: true },  age: { type: Number, min: 18 } });  // Create a model based on the schema const User = mongoose.model('User', userSchema);  // Export the model for use in other parts of the application module.exports = User; |
| --- |

1. **Create a Model:** Once you've defined a schema, you can create a model based on that schema using the mongoose.model() method. The first argument is the name of the model (which corresponds to the MongoDB collection name), and the second argument is the schema definition.
2. **Use the Model:** You can now use the model to perform CRUD (Create, Read, Update, Delete) operations on MongoDB documents. For example

**Sample Queries:**

* **Create a new document:**

| const newUser = new User({  name: 'John Doe',  email: 'john@example.com',  age: 30 });  newUser.save()  .then((result) => {  console.log('User created:', result);  })  .catch((error) => {  console.error('Error creating user:', error);  }); |
| --- |

* **Read documents:**

| **// Find all users User.find()  .then((users) => {  console.log('Users:', users);  })  .catch((error) => {  console.error('Error finding users:', error);  });  // Find a user by ID User.findById(userId)  .then((user) => {  console.log('User:', user);  })  .catch((error) => {  console.error('Error finding user:', error);  });** |
| --- |

* **Update documents:**

| **// Update a user by ID User.findByIdAndUpdate(userId, { age: 35 }, { new: true })  .then((updatedUser) => {  console.log('Updated user:', updatedUser);  })  .catch((error) => {  console.error('Error updating user:', error);  });** |
| --- |

* **Delete documents:**

| // Delete a user by ID User.findByIdAndDelete(userId)  .then(() => {  console.log('User deleted');  })  .catch((error) => {  console.error('Error deleting user:', error);  }); |
| --- |

**API Example**

To extend the previous REST API example to use a database, we'll integrate Mongoose, define a schema for users, and modify the routes to perform CRUD operations on the MongoDB database. Let's go through the steps:

1. Install Mongoose: If you have
2. n't already, install Mongoose in your Node.js project using npm:

| npm install mongoose |
| --- |

1. **Define a Mongoose Schema:** Create a Mongoose schema for the "User" collection. This schema defines the structure of user documents stored in the database.

| // models/User.js const mongoose = require('mongoose');  // Define a schema for user documents const userSchema = new mongoose.Schema({  name: { type: String, required: true },  email: { type: String, required: true, unique: true },  age: { type: Number, min: 18 } });  // Create a model based on the schema const User = mongoose.model('User', userSchema);  module.exports = User; |
| --- |

1. **Update Routes to Use Database:** Modify the routes to interact with the MongoDB database using Mongoose. For each route (GET, POST, PUT, DELETE), perform the corresponding CRUD operation on the "User" collection.

| // routes/users.js const express = require('express'); const router = express.Router(); const User = require('../models/User');  // GET /users - Get all users router.get('/', (req, res) => {  User.find()  .then(users => res.json(users))  .catch(error => res.status(500).json({ error: 'Internal Server Error' })); });  // GET /users/:id - Get user by ID router.get('/:id', (req, res) => {  const userId = req.params.id;  User.findById(userId)  .then(user => {  if (!user) {  return res.status(404).json({ error: 'User not found' });  }  res.json(user);  })  .catch(error => res.status(500).json({ error: 'Internal Server Error' })); });  // POST /users - Create a new user router.post('/', (req, res) => {  const newUser = new User(req.body);  newUser.save()  .then(user => res.status(201).json(user))  .catch(error => res.status(400).json({ error: 'Bad Request' })); });  // PUT /users/:id - Update user by ID router.put('/:id', (req, res) => {  const userId = req.params.id;  User.findByIdAndUpdate(userId, req.body, { new: true })  .then(user => {  if (!user) {  return res.status(404).json({ error: 'User not found' });  }  res.json(user);  })  .catch(error => res.status(400).json({ error: 'Bad Request' })); });  // DELETE /users/:id - Delete user by ID router.delete('/:id', (req, res) => {  const userId = req.params.id;  User.findByIdAndDelete(userId)  .then(user => {  if (!user) {  return res.status(404).json({ error: 'User not found' });  }  res.json(user);  })  .catch(error => res.status(400).json({ error: 'Bad Request' })); });  module.exports = router; |
| --- |

1. **Use Routes in Express App:** Update your Express app to use the user routes defined above.

| // app.js const express = require('express'); const bodyParser = require('body-parser'); const mongoose = require('mongoose'); const userRoutes = require('./routes/users');  const app = express(); const port = 3000;  // Middleware to parse JSON requests app.use(bodyParser.json());  // Connect to MongoDB mongoose.connect('mongodb://localhost:27017/mydatabase', {  useNewUrlParser: true,  useUnifiedTopology: true }) .then(() => {  console.log('Connected to MongoDB'); }) .catch(error => {  console.error('Error connecting to MongoDB:', error); });  // Use user routes app.use('/users', userRoutes);  // Start the server app.listen(port, () => {  console.log(`Server is running on port ${port}`); }); |
| --- |