**Notes: MongoDB and Mongoose (FreeCodeCamp Backend Development & APIs)**

**What are MongoDB and Mongoose?**

**MongoDB** is a NoSQL database that stores data as JSON-like documents instead of traditional SQL tables. Unlike relational databases, MongoDB stores all related data within a single record, making it more flexible for modern applications.

**Mongoose** is an npm package that provides an elegant MongoDB object modeling layer for Node.js. It simplifies working with MongoDB by allowing you to use plain JavaScript objects and provides features like schemas, validation, and query building.

**1. Setting Up MongoDB Atlas & Mongoose Connection**

**MongoDB Atlas Setup**

MongoDB Atlas provides a free cloud-hosted MongoDB database. Follow these steps:

1. **Create Account**: Visit [MongoDB Atlas](https://www.mongodb.com/atlas) and create free account
2. **Create Cluster**: Choose free tier (M0 Sandbox)
3. **Database Access**: Create database user with username/password
4. **Network Access**: Add IP address (0.0.0.0/0 for development)
5. **Get Connection String**: Click "Connect" → "Connect your application" → Copy URI

**Environment Variables Setup**

# .env file  
MONGO\_URI='mongodb+srv://username:password@cluster.mongodb.net/database\_name'

**Mongoose Connection Code**

const mongoose = require('mongoose');  
require('dotenv').config();  
  
const mongoURI = process.env.MONGO\_URI;  
  
// Check if URI exists  
if (!mongoURI) {  
 console.error("Error: MONGO\_URI is not defined in your .env file");  
 process.exit(1); // Stop the app if no URI is found  
}  
  
// Connect to MongoDB  
mongoose.connect(mongoURI, {  
 useNewUrlParser: true,  
 useUnifiedTopology: true,  
})  
.then(() => console.log("MongoDB connected successfully"))  
.catch(err => {  
 console.error("MongoDB connection error:", err);  
 process.exit(1);  
});

**Key Points:**

* useNewUrlParser: true - Uses new URL parser to avoid deprecation warnings
* useUnifiedTopology: true - Uses new topology engine for better connection handling
* .then()/.catch() - Handles connection success/failure with promises
* process.exit(1) - Terminates app if connection fails

**2. Creating Schemas and Models**

**Schema Definition**

Schemas define the structure and data types for documents in a collection.

const mongoose = require('mongoose');  
  
// Define schema structure  
const personSchema = new mongoose.Schema({  
 name: { type: String, required: true }, // Required string field  
 age: Number, // Optional number field  
 favoriteFoods: [String], // Array of strings  
});  
  
// Create model from schema  
const Person = mongoose.model('Person', personSchema);

**Schema Features:**

* **type**: Defines data type (String, Number, Date, Boolean, Array)
* **required**: Makes field mandatory
* **default**: Sets default value if none provided
* **unique**: Ensures field values are unique across collection
* **validate**: Custom validation functions

**3. CRUD Operations - CREATE**

**Creating and Saving Single Document**

const createAndSavePerson = (done) => {  
 // Create new document instance  
 const person = new Person({  
 name: 'John Doe',  
 age: 30,  
 favoriteFoods: ['Pizza', 'Burger'],  
 });  
  
 // Save to database  
 person.save((err, data) => {  
 if (err) {  
 console.error('Error saving person:', err);  
 return done(err); // Pass error to callback  
 }  
 console.log('Person saved:', data);  
 done(null, data); // Pass saved document to callback  
 });  
};

**How person.save() works:**

* Creates new document in MongoDB collection
* Validates data against schema rules
* Assigns unique \_id if not provided
* Returns saved document with MongoDB metadata
* Callback follows Node.js convention: (err, data)

**Creating Multiple Documents**

const createManyPeople = (arrayOfPeople, done) => {  
 // Use Model.create() to save array of objects  
 Person.create(arrayOfPeople, (err, data) => {  
 if (err) return done(err); // Handle errors  
 done(null, data); // Return array of created documents  
 });  
};

**How Person.create() works:**

* Takes array of objects as first parameter
* Creates multiple documents in single operation
* More efficient than multiple save() calls
* Returns array of created documents with \_id fields
* Validates each document against schema

**4. CRUD Operations - READ**

**Finding Multiple Documents**

const findPeopleByName = (personName, done) => {  
 // Search for all documents matching criteria  
 Person.find({ name: personName }, (err, data) => {  
 if (err) return done(err);  
 done(null, data); // Returns array (empty if no matches)  
 });  
};

**How Person.find() works:**

* First parameter: Query object { field: value }
* Returns array of matching documents
* Empty array [] if no matches found
* Can use complex queries with operators ($gt, $lt, $in, etc.)

**Finding Single Document**

const findOneByFood = (food, done) => {  
 // Find first document matching criteria  
 Person.findOne({ favoriteFoods: food }, (err, data) => {  
 if (err) return done(err);  
 done(null, data); // Returns single document or null  
 });  
};

**How Person.findOne() works:**

* Returns single document (not array)
* Returns null if no match found
* Useful when expecting only one result
* Stops searching after first match

**Finding by MongoDB ID**

const findPersonById = (personId, done) => {  
 // Search by MongoDB's unique \_id field  
 Person.findById(personId, (err, data) => {  
 if (err) return done(err);  
 done(null, data);  
 });  
};

**How Person.findById() works:**

* Optimized for searching by \_id field
* MongoDB automatically indexes \_id for fast lookups
* Returns single document or null
* personId can be string or ObjectId

**5. CRUD Operations - UPDATE**

**Find, Edit, Then Save Pattern**

const findEditThenSave = (personId, done) => {  
 const foodToAdd = "hamburger";  
  
 // First find the document  
 Person.findById(personId, (err, person) => {  
 if (err) return done(err);  
 if (!person) return done(new Error("Person not found"));  
   
 // Modify the document  
 person.favoriteFoods.push(foodToAdd);  
  
 // Save changes back to database  
 person.save((err, updatedPerson) => {  
 if (err) return done(err);  
 done(null, updatedPerson);  
 });  
 });  
};

**How this pattern works:**

* First retrieves document with findById()
* Modifies document in memory using JavaScript
* Array.push() adds element to favoriteFoods array
* save() persists changes to database
* Returns updated document

**Atomic Update Operations**

const findAndUpdate = (personName, done) => {  
 const ageToSet = 20;  
  
 Person.findOneAndUpdate(  
 { name: personName }, // Query criteria  
 { age: ageToSet }, // Update operations  
 { new: true }, // Options: return updated doc  
 (err, updatedPerson) => {  
 if (err) return done(err);  
 done(null, updatedPerson);  
 }  
 );  
};

**How findOneAndUpdate() works:**

* **Parameter 1**: Query object to find document
* **Parameter 2**: Update object with new values
* **Parameter 3**: Options object
  + { new: true }: Return updated document (default returns original)
  + { upsert: true }: Create if doesn't exist
  + { runValidators: true }: Run schema validators
* **Parameter 4**: Callback function
* Atomic operation (prevents race conditions)

**6. CRUD Operations - DELETE**

**Delete Single Document by ID**

const removeById = (personId, done) => {  
 Person.findByIdAndRemove(personId, (err, removedPerson) => {  
 if (err) return done(err);  
 done(null, removedPerson); // Returns the deleted document  
 });  
};

**How findByIdAndRemove() works:**

* Finds document by \_id and deletes it
* Returns the deleted document (not just confirmation)
* Returns null if document not found
* Alternative: findOneAndRemove() with custom query

**Delete Multiple Documents**

const removeManyPeople = (done) => {  
 const nameToRemove = "Mary";  
   
 Person.remove({ name: nameToRemove }, (err, result) => {  
 if (err) return done(err);  
 done(null, result); // Returns operation status, not documents  
 });  
};

**How Person.remove() works:**

* Deletes ALL documents matching query
* Returns operation result object: { n: 2, ok: 1 }
  + n: Number of documents deleted
  + ok: Success indicator (1 = success)
* **Warning**: remove() is deprecated, use deleteMany() in newer versions

**7. Query Chaining and Advanced Queries**

**Building Complex Queries**

const queryChain = (done) => {  
 const foodToSearch = "burrito";  
  
 Person.find({ favoriteFoods: foodToSearch }) // Find matching documents  
 .sort({ name: 1 }) // Sort by name (ascending)  
 .limit(2) // Limit to 2 results  
 .select('-age') // Exclude age field  
 .exec((err, data) => { // Execute query  
 if (err) return done(err);  
 done(null, data);  
 });  
};

**Query Chain Methods:**

* **.find(query)**: Initial query filter
* **.sort({ field: 1 })**: Sort results (1 = ascending, -1 = descending)
* **.limit(number)**: Limit number of results returned
* **.select('field -field')**: Include/exclude fields (- prefix excludes)
* **.skip(number)**: Skip first N results (pagination)
* **.exec(callback)**: Execute the built query

**Why use query chaining:**

* Builds query step by step without executing
* More readable for complex queries
* Can conditionally add filters
* Only executes when .exec() is called

**8. Error Handling and Callbacks**

**Node.js Callback Convention**

All Mongoose operations follow Node.js callback pattern:

function(err, data) {  
 if (err) {  
 // Handle error  
 return done(err);  
 }  
 // Handle success  
 done(null, data);  
}

**Common Error Types**

* **ValidationError**: Schema validation failed
* **CastError**: Data type conversion failed
* **MongoError**: Database connection or operation error
* **DocumentNotFoundError**: Document doesn't exist

**9. Complete Working Example**

require('dotenv').config();  
const mongoose = require('mongoose');  
  
// Database connection  
const mongoURI = process.env.MONGO\_URI;  
  
if (!mongoURI) {  
 console.error("Error: MONGO\_URI is not defined in your .env file");  
 process.exit(1);  
}  
  
mongoose.connect(mongoURI, {  
 useNewUrlParser: true,  
 useUnifiedTopology: true,  
})  
.then(() => console.log("MongoDB connected successfully"))  
.catch(err => {  
 console.error("MongoDB connection error:", err);  
 process.exit(1);  
});  
  
// Schema definition  
const personSchema = new mongoose.Schema({  
 name: { type: String, required: true },  
 age: Number,  
 favoriteFoods: [String],  
});  
  
// Model creation  
const Person = mongoose.model('Person', personSchema);  
  
// Create single document  
const createAndSavePerson = (done) => {  
 const person = new Person({  
 name: 'John Doe',  
 age: 30,  
 favoriteFoods: ['Pizza', 'Burger'],  
 });  
  
 person.save((err, data) => {  
 if (err) {  
 console.error('Error saving person:', err);  
 return done(err);  
 }  
 console.log('Person saved:', data);  
 done(null, data);  
 });  
};  
  
// Create multiple documents  
const createManyPeople = (arrayOfPeople, done) => {  
 Person.create(arrayOfPeople, (err, data) => {  
 if (err) return done(err);  
 done(null, data);  
 });  
};  
  
// Find multiple documents  
const findPeopleByName = (personName, done) => {  
 Person.find({name: personName}, (err, data) => {  
 if(err) return done(err);  
 done(null, data);  
 });  
};  
  
// Find single document by field  
const findOneByFood = (food, done) => {  
 Person.findOne({favoriteFoods: food}, (err, data) => {  
 if(err) return done(err);  
 done(null, data);  
 })  
};  
  
// Find by MongoDB ID  
const findPersonById = (personId, done) => {  
 Person.findById(personId, (err, data) => {  
 if(err) return done(err);  
 done(null, data);  
 })  
};  
  
// Classic update pattern  
const findEditThenSave = (personId, done) => {  
 const foodToAdd = "hamburger";  
  
 Person.findById(personId, (err, person) => {  
 if (err) return done(err);  
 if (!person) return done(new Error("Person not found"));  
   
 person.favoriteFoods.push(foodToAdd);  
  
 person.save((err, updatedPerson) => {  
 if (err) return done(err);  
 done(null, updatedPerson);  
 });  
 });  
};  
  
// Atomic update  
const findAndUpdate = (personName, done) => {  
 const ageToSet = 20;  
  
 Person.findOneAndUpdate(  
 { name: personName },  
 { age: ageToSet },  
 { new: true },  
 (err, updatedPerson) => {  
 if (err) return done(err);  
 done(null, updatedPerson);  
 }  
 );  
};  
  
// Delete single document  
const removeById = (personId, done) => {  
 Person.findByIdAndRemove(personId, (err, removedPerson) => {  
 if (err) return done(err);  
 done(null, removedPerson);  
 });  
};  
  
// Delete multiple documents  
const removeManyPeople = (done) => {  
 const nameToRemove = "Mary";  
 Person.remove({name: nameToRemove}, (err, data) => {  
 if(err) return done(err);  
 done(null, data);  
 })  
};  
  
// Query chaining  
const queryChain = (done) => {  
 const foodToSearch = "burrito";  
  
 Person.find({ favoriteFoods: foodToSearch })  
 .sort({ name: 1 })  
 .limit(2)  
 .select('-age')  
 .exec((err, data) => {  
 if (err) return done(err);  
 done(null, data);  
 });  
};  
  
// Export functions for testing  
exports.PersonModel = Person;  
exports.createAndSavePerson = createAndSavePerson;  
exports.findPeopleByName = findPeopleByName;  
exports.findOneByFood = findOneByFood;  
exports.findPersonById = findPersonById;  
exports.findEditThenSave = findEditThenSave;  
exports.findAndUpdate = findAndUpdate;  
exports.createManyPeople = createManyPeople;  
exports.removeById = removeById;  
exports.removeManyPeople = removeManyPeople;  
exports.queryChain = queryChain;

**10. Key Development Tips**

**Best Practices**

1. **Always handle errors**: Use proper error checking in callbacks
2. **Use schemas**: Define clear data structures with validation
3. **Environment variables**: Store connection strings securely
4. **Connection management**: Reuse single connection across app
5. **Query optimization**: Use indexes for frequently queried fields

**Common Pitfalls**

1. **Callback hell**: Use promises or async/await for complex operations
2. **Memory leaks**: Close database connections properly
3. **Validation errors**: Always validate data before saving
4. **Deprecated methods**: Replace remove() with deleteMany()

**Summary**

This module covers essential MongoDB and Mongoose concepts:

* **Database Setup**: Connecting to MongoDB Atlas cloud database
* **Schema Design**: Defining document structure with validation
* **CRUD Operations**: Create, Read, Update, Delete documents
* **Query Building**: Using find, update, and delete methods
* **Advanced Queries**: Chaining operations for complex searches
* **Error Handling**: Proper Node.js callback patterns

These skills form the foundation for building data-driven Node.js applications with persistent storage using MongoDB.