mongoose

PROF. P. M. JADAV
ASSOCIATE PROFESSOR,
COMPUTER ENGINEERING DEPARTMENT
FACULTY OF TECHNOLOGY
DHARMSINH DESAI UNIVERSITY, NADIAD

Content

- Introduction
- Mongoose Installation

Introduction

- Mongoose is an ODM (Object Data Modeling) library for MongoDB and Node.js
- Designed to work in an asynchronous environment
- Provides Schema-based solution to model our application data
- Includes built-in type casting, validation, query building, business logic hooks and more

Introduction

- Built on top of the official MongoDB Node.js driver
- Simplifies interactions with MongoDB collections

Why Use Mongoose?

- Enforces schema & validation in MongoDB (which is schema-less by default)
- Provides query building and data manipulation helpers
- Easy to define relationships between documents
- Middleware support (hooks for save, remove, etc.)
- Makes code more structured and maintainable

Installation

```
npm  install  mongoose
```

Connecting to MongoDB

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

// Connect and Create DB 'LibDB' if it doesn't exist

mongoose.connect('mongodb://127.0.0.1:27017/LibDB’)

   .then(() => console.log('connected!'))

   .catch(err => console.error('conn error:', err));
```

Data Types

- String
- Number
- Date
- Boolean
- Array
- Buffer
- Mixed
- ObjectId
- Decimal128
- Map

Data Type (String)

- Stores textual data
- Validators: required, minlength, maxlength, match (regex)

```
title: {
      type: String,
      required: true,
      minlength: 3
}
```

Data Type (Number)

- Stores numeric values (integers or floating point)
- Validators: min, max

```
price: {
    type: Number,
    min: 0,
    max: 1000
}
```

Data Type (Date)

- Stores date & time
- Defaults to current date using Date.now

```
publishedAt: { type: Date, default: Date.now }
```

Data Type (Boolean)

- Stores true or false

```
isPublished: {
    type: Boolean,
    default: false
}
```

Data Type (Array)

- Stores multiple values in an array
- Can hold simple types or subdocuments

```
genres: [String], // Array of strings

ratings: [{ score: Number, user: String }] // Array of objects
```

Data Type (Buffer)

- Stores binary data
  - e.g., images, files

```
coverImage: Buffer
```

Data Type (ObjectId)

- Special type for MongoDB document references
- Used for relationships between collections

```
author: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "User"
}
```

Data Type (Mixed)

- Allows any type of data to be stored
- Be wary of using this data type as it loses many advantages of Mongoose features, such as data validation and detecting entity changes to automatically know to update the property when saving

```
metadata: mongoose.Schema.Types.Mixed
```

Data Type (Decimal128)

- High-precision decimal values
- Better for financial data than Number

```
price: mongoose.Schema.Types.Decimal128
```

Data Type (Map)

- Stores key-value pairs (like JavaScript Map)
- Keys are strings, values follow specified type

```
extraInfo: { type: Map, of: String }
```

Schema (Nested/Sub-Document)

- Allows embedding objects inside documents

```
publisher: {
  name: String,
  location: String
}
```

Example

```
const bookSchema = new mongoose.Schema({
    title: { type: String, required: true },
    author: String,
    year: Number,
    price: mongoose.Schema.Types.Decimal128,
    inStock: { type: Boolean, default: true },
    genres: [String],
    publishedAt: { type: Date, default: Date.now },
    coverImage: Buffer,
```

Example (..continue)

```
    publisher: {
      name: String,
      location: String
    },
    reviews: [{
      user: { type: mongoose.Schema.Types.ObjectId,
                ref: "User" },
      comment: String,
      rating: Number
    }],
    metadata: mongoose.Schema.Types.Mixed,
    tags: { type: Map, of: String }
});
```

Defining a Schema

```
const bookSchema = new mongoose.Schema({
  title: String,
  author: String,
  year: Number,
  price: Number
});
```

- Schemas define the shape of documents
- Each field can have a type and validation rules

Models

```
const Book = mongoose.model("Book", bookSchema);
```

- A Model is a compiled version of the schema
- Provides an interface to interact with the database
- Maps to a MongoDB collection
  - pluralized form of model name → books

Queries

- Mongoose models provide several static helper functions for CRUD operations
- Each of these functions returns a mongoose Query object

```
Model.find()                       Model.findOneAndReplace()
Model.findById()                   Model.findOneAndUpdate()
Model.findByIdAndDelete()          Model.replaceOne()
Model.findByIdAndRemove()          Model.updateMany()
Model.findByIdAndUpdate()          Model.updateOne()
Model.findOne()                    Model.deleteMany()
Model.findOneAndDelete()           Model.deleteOne()
```

CRUD Operations

```
const newBook = new Book({ title: "test", author: "John Doe" });

await newBook.save();
```

Create

```
const books = await Book.find({ author: "John Doe" });
```

Read

```
await Book.updateOne({ title: "test" }, { price: 25 });
```

Update

```
await Book.deleteOne({ title: "Mongoose Basics" });
```

Delete

Built-In Validators

```
const bookSchema = new mongoose.Schema({
  title: { type: String, required: true, minlength: 3 },
  price: { type: Number, min: 0 },
  year:  { type: Number, max: new Date().getFullYear() }
});
```

Custom Validators

```
username: {
  type: String,
  required: true,
  validate: {
    validator: function(v) {
      return /^[a-zA-Z0-9_]{3,15}$/.test(v);
      // only letters, numbers, underscores, 3–15 chars
    },
    message: props => `${props.value} is not a valid username`
  }
}
```

Props

- When a validator fails, Mongoose passes an object called props into the message function
- This object contains useful details about the validation failure

Property	Description	Example
props.value	The actual value that failed validation	"bad-username!!"
props.path	The name of the field being validated	"username"
props.kind	Type of validation (e.g., "user defined", "required")	"user defined"
props.reason	If async validation failed, the error object	Error: Title already exists
props.validator	The validator function that failed	[Function: validator]

Relationships (Referencing Documents)

```
const userSchema = new mongoose.Schema({
  name: String,
  books:   [{
              type: mongoose.Schema.Types.ObjectId,
              ref: "Book"
          }]
});
```

Middleware (Hooks)

- Pre/Post hooks allow logic before/after actions

```
bookSchema.pre("save", function(next) {
  console.log("About to save:", this.title);
  next();
});
```

Aggregation

```
Book.aggregate([
  { $match: {
                price: { $gt: 20 }
             }
  },
  { $group: {
                _id: "$author",
                totalBooks: { $sum: 1 }
             }
  }
]);
```

Advantages

- Schema enforcement on schema-less MongoDB
- Simplified CRUD operations
- Middleware for business logic
- Validation support
- Relationships

Limitations

- Adds abstraction layer → slightly slower than native driver
- Less flexibility for very dynamic/no-schema data
- Learning curve for advanced features

Use Cases

- Applications needing structured data
- APIs requiring validation and consistency
- Projects with relationships
  - (Users ↔ Books, Orders ↔ Products)
- Systems that benefit from middleware hooks

References

- https://mongoosejs.com/docs/