Step 2: Todo App with Mongoose

Using ODM (Object Document Mapping)

What Changed?

- Step 1 used native MongoDB driver - direct database operations.
- Step 2 uses **Mongoose ODM** - adds a data modeling layer.
- Same functionality, but better code organization and data validation.
- This is about **software engineering improvement**, not adding features.

What is ODM?

Object Document Mapping (ODM)

- Maps JavaScript objects to MongoDB documents automatically.
- Similar to ORM (Object-Relational Mapping) for SQL databases.
- Provides schema definition, validation, and query building.
- Makes code more maintainable and less error-prone.

Key Improvements

1. **Schema Definition** - Explicit data structure
2. **Data Validation** - Built-in type checking
3. **Cleaner Code** - Model methods instead of raw queries
4. **Better Organization** - Separate model files
5. **Error Handling** - Automatic validation errors

Directory Structure

```
.
├── index.js
├── models
│   ├── Counter.js          # Counter schema for auto-increment
│   └── Post.js             # Post schema and model
├── util
│   ├── uri.js
│   └── util.js
└── views
    ├── detail.ejs
    ├── edit.ejs
    ├── list.ejs
    ├── nav.ejs
    └── write.ejs
```

New: `models/` directory for schema definitions

Data Model - Before (Step 1)

In `util/util.js`:

```
const newPost = {
  _id: newId,
  title: req.body.title,
  date: req.body.date
};
```

- Just a plain JavaScript object
- No type checking
- No validation
- Structure defined inline

Data Model - After (Step 2)

In `models/Post.js`:

```
const postSchema = new mongoose.Schema({
  _id: {
    type: Number,
    required: true,
  },
  title: {
    type: String,
    required: true,
  },
  date: {
    type: String,
    required: true,
  },
});

const Post = mongoose.model('Post', postSchema, 'posts');
```

Schema Benefits

1. **Type Safety** - `type: Number`, `type: String`
2. **Validation** - `required: true`
3. **Documentation** - Schema shows data structure
4. **Reusability** - Import model anywhere
5. **Single Source of Truth** - One place defines structure

Database Connection - Before

```
import { MongoClient } from 'mongodb';

const client = new MongoClient(uri);
await client.connect();
db = client.db(DATABASE);
const posts = db.collection('posts');
```

- Manual connection setup
- Access collections directly
- More boilerplate code

Database Connection - After

```
import mongoose from 'mongoose';

async function connectDB() {
  try {
    await mongoose.connect(uri);
    console.log('MongoDB connected successfully');
  } catch (err) {
    console.error('MongoDB connection error:', err);
    process.exit(1);
  }
}

await connectDB();
```

- Simpler connection
- Built-in connection pooling
- Automatic reconnection handling

CRUD Operations - Create

Before:

```
await posts.insertOne({
  _id: newId,
  title: req.body.title,
  date: req.body.date
});
```

After:

```
const newPost = new Post({
  _id: nextId,
  title: req.body.title,
  date: req.body.date,
});

await newPost.save();
```

CRUD Operations - Read

Before:

```
const posts = await posts.find({}).toArray();
const post = await posts.findOne({ _id: parseInt(id) });
```

After:

```
const posts = await Post.find({});
const post = await Post.findById(id);
```

- More intuitive method names
- No need for `toArray()`
- `findById()` is clearer than `findOne({ _id: ... })`

CRUD Operations - Update

Before:

```
await posts.updateOne(
  { _id: parseInt(req.body.id) },
  { $set: { title: req.body.title, date: req.body.date } }
);
```

After:

```
await Post.findByIdAndUpdate(
  postId,
  {
    title: req.body.title,
    date: req.body.date,
  }
);
```

- No need for `$set` operator

CRUD Operations - Delete

Before:

```
await posts.deleteOne({ _id: parseInt(req.body._id) });
```

After:

```
const postId = parseInt(req.body._id);
await Post.findByIdAndDelete(postId);
```

- Self-documenting method name
- Type conversion separated from query

Counter Model

```
const counterSchema = new mongoose.Schema({
  _id: {
    type: String,
    required: true,
  },
  seq: {
    type: Number,
    default: 0,
  },
});

const Counter = mongoose.model('Counter', counterSchema, 'counter');
```

- Separate model for auto-incrementing IDs
- Demonstrates schema reusability

Auto-Increment - Before

```
const result = await counter.findOneAndUpdate(
  { _id: "postId" },
  { $inc: { seq: 1 } },
  { returnDocument: "after", upsert: true }
);
return result.seq;
```

Auto-Increment - After

```
export async function getNextId() {
  const result = await Counter.findByIdAndUpdate(
    'postId',
    { $inc: { seq: 1 } },
    {
      new: true,          // Return updated document
      upsert: true,       // Create if doesn't exist
    }
  );

  return result.seq;
}
```

- `new: true` more intuitive than `returnDocument: "after"`
- Better parameter names

Running Application

Install packages and run:

```
npm install
npm start
```

Or with nodemon for auto-restart:

```
npm install -g nodemon
nodemon ./index.js
```

Same commands as Step 1!

When to Use Native vs Mongoose?

Use Native MongoDB Driver:

- Maximum performance critical
- Very dynamic schemas
- Full control over queries needed

Use Mongoose ODM:

- Standard CRUD applications
- Team collaboration (schemas document data)
- Want validation and type safety
- Middleware/hooks needed

Software Engineering Benefits

1. **Maintainability** - Clear data models in separate files
2. **Scalability** - Easy to add validation rules
3. **Team Collaboration** - Schemas serve as documentation
4. **Error Prevention** - Type validation catches bugs early
5. **Code Quality** - More readable and self-documenting

Key Concepts

- **ODM (Object Document Mapping)** - Maps objects to documents
- **Schema** - Defines data structure and validation
- **Model** - Interface for database operations
- **Separation of Concerns** - Models in separate directory
- **Code Quality** - Same features, better implementation

Consider the Trade-Offs

- We add complexity with Mongoose for better software engineering.
- Is the trade-off worth it for your project?
- Always evaluate based on project needs and team skills.