Complete Express.js REST API Guide with MongoDB & Mongoose

Table of Contents

What is Express.js?

Express.js is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. It's designed for building web applications and APIs quickly and efficiently.

Key Features:

- Fast, unopinionated, minimalist web framework
- Robust routing system
- Middleware support
- Template engine support
- Static file serving
- Error handling

REST API Fundamentals

REST (Representational State Transfer) is an architectural style for designing networked applications. A RESTful API uses HTTP methods to perform CRUD operations.

HTTP Methods:

- **GET**: Retrieve data
- **POST**: Create new data

- **PUT**: Update existing data (complete replacement)
- **PATCH**: Update existing data (partial update)
- **DELETE**: Remove data

REST Principles:

1. **Stateless**: Each request contains all information needed
2. **Client-Server**: Separation of concerns
3. **Cacheable**: Responses should be cacheable when appropriate
4. **Uniform Interface**: Consistent resource identification
5. **Layered System**: Architecture can be composed of hierarchical layers

Example REST Endpoints:

```
GET    /api/users          # Get all users
GET    /api/users/:id      # Get specific user
POST   /api/users          # Create new user
PUT    /api/users/:id      # Update entire user
PATCH  /api/users/:id      # Update part of user
DELETE /api/users/:id      # Delete user
```

MVC Architecture

MVC (Model-View-Controller) is a software architectural pattern that separates an application into three interconnected components.

Components:

Model

- Represents data and business logic
- Handles data validation
- Interacts with the database
- Independent of user interface

View

- Presents data to the user
- In APIs, this is typically JSON responses
- Handles data formatting and presentation

Controller

- Acts as intermediary between Model and View
- Handles user input and requests
- Contains application logic
- Decides which model to call and which view to render

Benefits of MVC:

- **Separation of Concerns**: Each component has a specific responsibility
- **Reusability**: Components can be reused across different parts
- **Maintainability**: Easier to maintain and update
- **Testability**: Each component can be tested independently

Setting Up Express.js

Installation

bash

```bash
# Initialize new Node.js project
npm init -y

# Install Express.js
npm install express

# Install development dependencies
npm install -D nodemon

# Install additional packages for complete setup
npm install mongoose cors helmet morgan dotenv bcryptjs jsonwebtoken
```

Basic Express Server

javascript

```javascript
// server.js
const express = require('express');
const app = express();
const PORT = process.env.PORT || 3000;

// Basic middleware
app.use(express.json());
app.use(express.urlencoded({ extended: true }));

// Basic route
app.get('/', (req, res) => {
  res.json({ message: 'Welcome to Express API' });
});

app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

Package.json Scripts

```json
{
  "scripts": {
    "start": "node server.js",
    "dev": "nodemon server.js"
  }
}
```

MongoDB Connection

MongoDB is a NoSQL document database. Here's how to connect Express.js to MongoDB.

Using MongoDB Driver

```javascript
```

```javascript
// config/database.js
const { MongoClient } = require('mongodb');

const connectDB = async () => {
  try {
    const client = new MongoClient(process.env.MONGODB_URI);
    await client.connect();
    console.log('MongoDB connected successfully');
    return client.db('your_database_name');
  } catch (error) {
    console.error('MongoDB connection error:', error);
    process.exit(1);
  }
};

module.exports = connectDB;
```

Using Mongoose (Recommended)

javascript

```javascript
// config/database.js
const mongoose = require('mongoose');

const connectDB = async () => {
  try {
    const conn = await mongoose.connect(process.env.MONGODB_URI, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });
    console.log(`MongoDB Connected: ${conn.connection.host}`);
  } catch (error) {
    console.error('Database connection error:', error);
    process.exit(1);
  }
};

module.exports = connectDB;
```

Environment Variables (.env)

```
MONGODB_URI=mongodb://localhost:27017/your_database_name
# For MongoDB Atlas
# MONGODB_URI=mongodb+srv://username:password@cluster.mongodb.net/database_name
PORT=3000
JWT_SECRET=your_jwt_secret_key
```

Mongoose ODM

Mongoose is an Object Document Mapper (ODM) for MongoDB and Node.js. It provides a straightforward schema-based solution to model application data.

Key Features:

- Schema definitions
- Data validation
- Middleware (hooks)
- Query building
- Type casting
- Built-in validators

Creating Schemas and Models

User Schema Example

javascript

```
// models/User.js
const mongoose = require('mongoose');
const bcrypt = require('bcryptjs');

const userSchema = new mongoose.Schema({
  username: {
    type: String,
    required: [true, 'Username is required'],
    unique: true,
    trim: true,
    minlength: [3, 'Username must be at least 3 characters'],
    maxlength: [20, 'Username cannot exceed 20 characters']
  },
  email: {
    type: String,
    required: [true, 'Email is required'],
    unique: true,
    lowercase: true,
    match: [/^\w+([.-]?\w+)*@\w+([.-]?\w+)*(\.\w{2,3})+$/, 'Please enter a valid email']
  },
  password: {
    type: String,
    required: [true, 'Password is required'],
    minlength: [6, 'Password must be at least 6 characters'],
    select: false // Don't include password in queries by default
  },
  role: {
    type: String,
    enum: ['user', 'admin'],
    default: 'user'
  },
  profile: {
    firstName: String,
    lastName: String,
    age: {
      type: Number,
      min: [0, 'Age cannot be negative'],
      max: [150, 'Age seems unrealistic']
    },
    avatar: String
  },
  isActive: {
    type: Boolean,
    default: true
  }
}, {
```

```
  timestamps: true, // Adds createdAt and updatedAt fields
  toJSON: { virtuals: true },
  toObject: { virtuals: true }
});

// Virtual field
userSchema.virtual('fullName').get(function() {
  return `${this.profile.firstName} ${this.profile.lastName}`;
});

// Pre-save middleware to hash password
userSchema.pre('save', async function(next) {
  if (!this.isModified('password')) return next();

  this.password = await bcrypt.hash(this.password, 12);
  next();
});

// Instance method to compare passwords
userSchema.methods.comparePassword = async function(candidatePassword) {
  return await bcrypt.compare(candidatePassword, this.password);
};

// Static method
userSchema.statics.findByEmail = function(email) {
  return this.findOne({ email });
};

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

Product Schema Example

javascript

```
// models/Product.js
const mongoose = require('mongoose');

const productSchema = new mongoose.Schema({
  name: {
    type: String,
    required: [true, 'Product name is required'],
    trim: true,
    maxlength: [100, 'Product name cannot exceed 100 characters']
  },
  description: {
    type: String,
    required: [true, 'Product description is required'],
    maxlength: [2000, 'Description cannot exceed 2000 characters']
  },
  price: {
    type: Number,
    required: [true, 'Product price is required'],
    min: [0, 'Price cannot be negative']
  },
  category: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Category',
    required: [true, 'Product category is required']
  },
  tags: [String],
  images: [String],
  inStock: {
    type: Boolean,
    default: true
  },
  quantity: {
    type: Number,
    default: 0,
    min: [0, 'Quantity cannot be negative']
  },
  ratings: {
    average: {
      type: Number,
      default: 0,
      min: 0,
      max: 5
    },
    count: {
      type: Number,
      default: 0
```

```
    }
  }
}, {
  timestamps: true
});

// Index for better query performance
productSchema.index({ name: 'text', description: 'text' });
productSchema.index({ category: 1, price: 1 });

module.exports = mongoose.model('Product', productSchema);
```

Mongoose Query Methods

javascript

```
// Basic CRUD operations
const User = require('./models/User');

// Create
const newUser = await User.create({
  username: 'johndoe',
  email: 'john@example.com',
  password: 'password123'
});

// Read
const users = await User.find(); // Get all users
const user = await User.findById(userId); // Get by ID
const user = await User.findOne({ email: 'john@example.com' }); // Get by field

// Update
const updatedUser = await User.findByIdAndUpdate(
  userId,
  { username: 'newusername' },
  { new: true, runValidators: true }
);

// Delete
await User.findByIdAndDelete(userId);

// Advanced queries
const users = await User.find({ isActive: true })
  .select('username email')
  .sort({ createdAt: -1 })
  .limit(10)
  .skip(20);

// Population (joining collections)
const products = await Product.find()
  .populate('category', 'name description')
  .exec();
```

Middleware in Express

Middleware functions are functions that have access to the request object (req), response object (res), and the next middleware function in the application's request-response cycle.

Types of Middleware:

1. **Application-level middleware**
2. **Router-level middleware**

3. **Error-handling middleware**
4. **Built-in middleware**
5. **Third-party middleware**

Built-in Middleware

javascript

```javascript
// Parse JSON bodies
app.use(express.json());

// Parse URL-encoded bodies
app.use(express.urlencoded({ extended: true }));

// Serve static files
app.use(express.static('public'));
```

Third-party Middleware

javascript

```javascript
const cors = require('cors');
const helmet = require('helmet');
const morgan = require('morgan');

// Enable CORS
app.use(cors());

// Security middleware
app.use(helmet());

// HTTP request logger
app.use(morgan('combined'));
```

Application-level Middleware

javascript

```
// Logger middleware
app.use((req, res, next) => {
  console.log(`${req.method} ${req.path} - ${new Date().toISOString()}`);
  next();
});

// Authentication middleware
const authenticate = (req, res, next) => {
  const token = req.header('Authorization')?.replace('Bearer ', '');

  if (!token) {
    return res.status(401).json({ error: 'Access token is required' });
  }

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch (error) {
    res.status(401).json({ error: 'Invalid token' });
  }
};
```

Creating Custom Middleware

Authentication Middleware

javascript

```javascript
// middleware/auth.js
const jwt = require('jsonwebtoken');
const User = require('../models/User');

const authenticate = async (req, res, next) => {
  try {
    const token = req.header('Authorization')?.replace('Bearer ', '');

    if (!token) {
      return res.status(401).json({
        success: false,
        message: 'Access token is required'
      });
    }

    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    const user = await User.findById(decoded.id);

    if (!user) {
      return res.status(401).json({
        success: false,
        message: 'User not found'
      });
    }

    req.user = user;
    next();
  } catch (error) {
    res.status(401).json({
      success: false,
      message: 'Invalid token'
    });
  }
};

module.exports = authenticate;
```

Authorization Middleware

```javascript
```

```javascript
// middleware/authorize.js
const authorize = (...roles) => {
  return (req, res, next) => {
    if (!roles.includes(req.user.role)) {
      return res.status(403).json({
        success: false,
        message: 'Access denied. Insufficient permissions'
      });
    }
    next();
  };
};

module.exports = authorize;
```

Validation Middleware

javascript

```javascript
// middleware/validation.js
const { body, validationResult } = require('express-validator');

const validateUser = [
  body('username')
    .isLength({ min: 3, max: 20 })
    .withMessage('Username must be between 3 and 20 characters')
    .isAlphanumeric()
    .withMessage('Username must contain only letters and numbers'),

  body('email')
    .isEmail()
    .withMessage('Please provide a valid email')
    .normalizeEmail(),

  body('password')
    .isLength({ min: 6 })
    .withMessage('Password must be at least 6 characters long')
    .matches(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)/)
    .withMessage('Password must contain at least one uppercase letter, one lowercase letter, and one number'),

  (req, res, next) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({
        success: false,
        message: 'Validation failed',
        errors: errors.array()
      });
    }
    next();
  }
];

module.exports = { validateUser };
```

Error Handling Middleware

```javascript
```

```
// middleware/errorHandler.js
const errorHandler = (err, req, res, next) => {
  let error = { ...err };
  error.message = err.message;

  // Log error
  console.log(err);

  // Mongoose bad ObjectId
  if (err.name === 'CastError') {
    const message = 'Resource not found';
    error = { message, statusCode: 404 };
  }

  // Mongoose duplicate key
  if (err.code === 11000) {
    const message = 'Duplicate field value entered';
    error = { message, statusCode: 400 };
  }

  // Mongoose validation error
  if (err.name === 'ValidationError') {
    const message = Object.values(err.errors).map(val => val.message);
    error = { message, statusCode: 400 };
  }

  res.status(error.statusCode || 500).json({
    success: false,
    error: error.message || 'Server Error'
  });
};

module.exports = errorHandler;
```

Building a Complete REST API

Project Structure

```
project/
├── server.js
├── config/
│   └── database.js
├── models/
│   ├── User.js
│   └── Product.js
├── controllers/
│   ├── authController.js
│   ├── userController.js
│   └── productController.js
├── routes/
│   ├── auth.js
│   ├── users.js
│   └── products.js
├── middleware/
│   ├── auth.js
│   ├── authorize.js
│   ├── validation.js
│   └── errorHandler.js
└── utils/
    └── asyncHandler.js
```

Async Handler Utility

javascript

```javascript
// utils/asyncHandler.js
const asyncHandler = (fn) => (req, res, next) =>
  Promise.resolve(fn(req, res, next)).catch(next);

module.exports = asyncHandler;
```

Controllers

Auth Controller

javascript

```
// controllers/authController.js
const asyncHandler = require('../utils/asyncHandler');
const User = require('../models/User');
const jwt = require('jsonwebtoken');

// Generate JWT Token
const generateToken = (id) => {
  return jwt.sign({ id }, process.env.JWT_SECRET, {
    expiresIn: process.env.JWT_EXPIRE || '30d'
  });
};

// @desc    Register user
// @route   POST /api/auth/register
// @access  Public
const register = asyncHandler(async (req, res) => {
  const { username, email, password } = req.body;

  // Create user
  const user = await User.create({
    username,
    email,
    password
  });

  // Create token
  const token = generateToken(user._id);

  res.status(201).json({
    success: true,
    token,
    data: {
      id: user._id,
      username: user.username,
      email: user.email,
      role: user.role
    }
  });
});

// @desc    Login user
// @route   POST /api/auth/login
// @access  Public
const login = asyncHandler(async (req, res) => {
  const { email, password } = req.body;
```

```
  // Validate email & password
  if (!email || !password) {
    return res.status(400).json({
      success: false,
      message: 'Please provide email and password'
    });
  }

  // Check for user
  const user = await User.findOne({ email }).select('+password');

  if (!user) {
    return res.status(401).json({
      success: false,
      message: 'Invalid credentials'
    });
  }

  // Check if password matches
  const isMatch = await user.comparePassword(password);

  if (!isMatch) {
    return res.status(401).json({
      success: false,
      message: 'Invalid credentials'
    });
  }

  // Create token
  const token = generateToken(user._id);

  res.status(200).json({
    success: true,
    token,
    data: {
      id: user._id,
      username: user.username,
      email: user.email,
      role: user.role
    }
  });
});

module.exports = {
  register,
```

```
  login
};
```

User Controller

javascript

```
// controllers/userController.js
const asyncHandler = require('../utils/asyncHandler');
const User = require('../models/User');

// @desc    Get all users
// @route   GET /api/users
// @access  Private/Admin
const getUsers = asyncHandler(async (req, res) => {
  const page = parseInt(req.query.page) || 1;
  const limit = parseInt(req.query.limit) || 10;
  const skip = (page - 1) * limit;

  const users = await User.find()
    .select('-password')
    .limit(limit)
    .skip(skip)
    .sort({ createdAt: -1 });

  const total = await User.countDocuments();

  res.status(200).json({
    success: true,
    count: users.length,
    pagination: {
      page,
      limit,
      total,
      pages: Math.ceil(total / limit)
    },
    data: users
  });
});

// @desc    Get single user
// @route   GET /api/users/:id
// @access  Private
const getUser = asyncHandler(async (req, res) => {
  const user = await User.findById(req.params.id);

  if (!user) {
    return res.status(404).json({
      success: false,
      message: 'User not found'
    });
  }
```

```
  res.status(200).json({
    success: true,
    data: user
  });
});

// @desc    Update user
// @route   PUT /api/users/:id
// @access  Private
const updateUser = asyncHandler(async (req, res) => {
  let user = await User.findById(req.params.id);

  if (!user) {
    return res.status(404).json({
      success: false,
      message: 'User not found'
    });
  }

  // Make sure user is owner or admin
  if (user._id.toString() !== req.user.id && req.user.role !== 'admin') {
    return res.status(401).json({
      success: false,
      message: 'Not authorized to update this user'
    });
  }

  user = await User.findByIdAndUpdate(req.params.id, req.body, {
    new: true,
    runValidators: true
  });

  res.status(200).json({
    success: true,
    data: user
  });
});

// @desc    Delete user
// @route   DELETE /api/users/:id
// @access  Private/Admin
const deleteUser = asyncHandler(async (req, res) => {
  const user = await User.findById(req.params.id);

  if (!user) {
    return res.status(404).json({
      success: false,
```

```
      message: 'User not found'
    });
  }

  await user.remove();

  res.status(200).json({
    success: true,
    data: {}
  });
});

module.exports = {
  getUsers,
  getUser,
  updateUser,
  deleteUser
};
```

Routes

Auth Routes

javascript

```
// routes/auth.js
const express = require('express');
const { register, login } = require('../controllers/authController');
const { validateUser } = require('../middleware/validation');

const router = express.Router();

router.post('/register', validateUser, register);
router.post('/login', login);

module.exports = router;
```

User Routes

javascript

```javascript
// routes/users.js
const express = require('express');
const {
  getUsers,
  getUser,
  updateUser,
  deleteUser
} = require('../controllers/userController');

const authenticate = require('../middleware/auth');
const authorize = require('../middleware/authorize');

const router = express.Router();

router.use(authenticate); // Protect all routes

router
  .route('/')
  .get(authorize('admin'), getUsers);

router
  .route('/:id')
  .get(getUser)
  .put(updateUser)
  .delete(authorize('admin'), deleteUser);

module.exports = router;
```

Complete Server Setup

javascript

```
// server.js
const express = require('express');
const dotenv = require('dotenv');
const cors = require('cors');
const helmet = require('helmet');
const morgan = require('morgan');
const connectDB = require('./config/database');
const errorHandler = require('./middleware/errorHandler');

// Load env vars
dotenv.config();

// Connect to database
connectDB();

const app = express();

// Body parser middleware
app.use(express.json());
app.use(express.urlencoded({ extended: false }));

// Security middleware
app.use(helmet());
app.use(cors());

// Logging middleware
if (process.env.NODE_ENV === 'development') {
  app.use(morgan('dev'));
}

// Route files
const auth = require('./routes/auth');
const users = require('./routes/users');

// Mount routers
app.use('/api/auth', auth);
app.use('/api/users', users);

// Error handler
app.use(errorHandler);

const PORT = process.env.PORT || 3000;

const server = app.listen(PORT, () => {
  console.log(`Server running in ${process.env.NODE_ENV} mode on port ${PORT}`);
});
```

```
// Handle unhandled promise rejections
process.on('unhandledRejection', (err, promise) => {
  console.log(`Error: ${err.message}`);
  // Close server & exit process
  server.close(() => {
    process.exit(1);
  });
});
```

Best Practices

1. Project Structure

- Organize code into logical directories
- Separate concerns (models, controllers, routes, middleware)
- Use meaningful file and folder names

2. Error Handling

- Use try-catch blocks or async handlers
- Create centralized error handling middleware
- Provide meaningful error messages
- Log errors for debugging

3. Validation

- Validate input data on both client and server
- Use schema validation (Mongoose) and request validation
- Sanitize user input to prevent injection attacks

4. Security

- Use HTTPS in production
- Implement proper authentication and authorization
- Use security middleware (helmet, cors)
- Validate and sanitize all inputs
- Use environment variables for sensitive data

5. Performance

- Use indexes in MongoDB for better query performance
- Implement pagination for large datasets

- Use caching where appropriate
- Optimize database queries

6. Documentation

- Document your API endpoints
- Use tools like Swagger/OpenAPI
- Write clear comments in code
- Maintain README files

7. Testing

- Write unit tests for controllers and models
- Implement integration tests for API endpoints
- Use testing frameworks like Jest or Mocha

8. Environment Management

- Use different configurations for development, testing, and production
- Use environment variables for configuration
- Never commit sensitive data to version control

This comprehensive guide covers all the essential aspects of building REST APIs with Express.js, MongoDB, and Mongoose. Each section provides practical examples and best practices to help you build robust and scalable applications.