Blog App Backend — Full Solution (Task-wise)

Includes project setup, full working backend (MongoDB + JWT auth), routes, and usage examples.

Table of Contents

Task 1 — Project Initialization & Dependencies

Explanation:

Create a new Node.js project and install required dependencies. We'll use Express for the server, Mongoose for MongoDB, bcryptjs for password hashing, jsonwebtoken for JWT auth, dotenv for environment variables, and cors for cross-origin requests. Nodemon is recommended for development.

Commands:

```
mkdir blog-backend
cd blog-backend
npm init -y
npm i express mongoose bcryptjs jsonwebtoken dotenv cors
npm i -D nodemon
# Add scripts in package.json:
# "start": "node index.js",
# "dev": "nodemon index.js"
```

Task 2 — Environment Configuration (.env)

Explanation:

Create a .env file in the project root. This file contains configuration that should NOT be committed to version control (add .env to .gitignore). Replace values as needed; if you use MongoDB Atlas, use the provided connection string.

.env example:

```
PORT=3000
MONGO_URI=mongodb://localhost:27017/blogdb
# If using Atlas, MONGO_URI=mongodb+srv://<user>:<pass>@cluster0.mongodb.net/blogdb?retryWrites=true&w=majorit
JWT_SECRET=your_jwt_secret_here
JWT_EXPIRES_IN=7d
```

Task 3 — Entry Point (index.js) & Server

Explanation:

index.js sets up the Express app, middleware, routes, and connects to MongoDB using Mongoose. A root route is included to verify the server is running.

index.js (complete):

```
require('dotenv').config();
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');

const authRoutes = require('./routes/auth');
const blogRoutes = require('./routes/blogs');

const app = express();
app.use(cors());
app.use(express.json());

// Root route for quick check
app.get('/', (req, res) => {
  res.send('Blog backend server running');
});

// Mount routes
app.use('/auth', authRoutes);
app.use('/blogs', blogRoutes);

// Connect to MongoDB and start server
const PORT = process.env.PORT || 3000;
mongoose.connect(process.env.MONGO_URI, {
  useNewUrlParser: true, useUnifiedTopology: true
}).then(() => {
  console.log('Connected to MongoDB');
  app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
}).catch(err => {
  console.error('MongoDB connection error:', err);
});
```

Task 4 — Models (User, Blog)

Explanation:

Define two Mongoose models: User and Blog. User stores username and hashed password. Blog stores title, description, and a reference to the author (User ObjectId).

models/User.js:

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

const userSchema = new mongoose.Schema({
  username: { type: String, required: true, unique: true },
  password: { type: String, required: true } // will store hashed password
}, { timestamps: true });

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

models/Blog.js:

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

const blogSchema = new mongoose.Schema({
  title: { type: String, required: true },
  description: { type: String, required: true },
  author: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true }
}, { timestamps: true });

module.exports = mongoose.model('Blog', blogSchema);
```

Task 5 — Authentication Middleware

Explanation:

Middleware verifies the JWT sent in Authorization header (Bearer token). If valid, it attaches the user (without password) to req.user and calls next().

middleware/auth.js:

```
const jwt = require('jsonwebtoken');
const User = require('../models/User');

module.exports = async function (req, res, next) {
  const authHeader = req.headers.authorization || '';
  const token = authHeader.startsWith('Bearer ') ? authHeader.slice(7) : null;
  if (!token) return res.status(401).json({ message: 'No token provided' });

  try {
    const payload = jwt.verify(token, process.env.JWT_SECRET);
    const user = await User.findById(payload.id).select('-password');
    if (!user) return res.status(401).json({ message: 'Invalid token' });
    req.user = user;
    next();
  } catch (err) {
    return res.status(401).json({ message: 'Invalid token', error: err.message });
  }
};
```

Task 6 — Routes: Auth (signup, signin)

Explanation:

Auth routes allow users to sign up and sign in. Passwords are hashed with bcryptjs. On signin, a JWT is issued containing the user's id.

routes/auth.js:

```
const express = require('express');
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');
const User = require('../models/User');

const router = express.Router();

// Signup
router.post('/signup', async (req, res) => {
  try {
    const { username, password } = req.body;
    if (!username || !password) return res.status(400).json({ message: 'username and password required' });

    const exist = await User.findOne({ username });
    if (exist) return res.status(400).json({ message: 'Username already exists' });

    const salt = await bcrypt.genSalt(10);
    const hashed = await bcrypt.hash(password, salt);
    const user = await User.create({ username, password: hashed });

    res.status(201).json({ message: 'User created', user: { id: user._id, username: user.username }});
  } catch (err) {
    res.status(500).json({ message: 'Error', error: err.message });
  }
});

// Signin
router.post('/signin', async (req, res) => {
  try {
    const { username, password } = req.body;
    if (!username || !password) return res.status(400).json({ message: 'username and password required' });

    const user = await User.findOne({ username });
    if (!user) return res.status(400).json({ message: 'Invalid credentials' });

    const match = await bcrypt.compare(password, user.password);
    if (!match) return res.status(400).json({ message: 'Invalid credentials' });

    const token = jwt.sign({ id: user._id, username: user.username }, process.env.JWT_SECRET, { expiresIn: pro

    res.json({ message: 'Signin success', token });
  } catch (err) {
    res.status(500).json({ message: 'Error', error: err.message });
  }
});

module.exports = router;
```

Task 7 — Routes: Blogs (CRUD)

Explanation:

Public endpoint to list blogs. Creating, updating, and deleting blogs require authentication. Only the author can update or delete their blog.

routes/blogs.js:

```
const express = require('express');
const Blog = require('../models/Blog');
const auth = require('../middleware/auth');

const router = express.Router();

// GET /blogs - public
router.get('/', async (req, res) => {
  try {
    const blogs = await Blog.find().populate('author', 'username').sort({ createdAt: -1 });
    res.json(blogs);
  } catch (err) {
    res.status(500).json({ message: 'Error', error: err.message });
  }
});

// POST /blogs - create (authenticated)
router.post('/', auth, async (req, res) => {
  try {
    const { title, description } = req.body;
    if (!title || !description) return res.status(400).json({ message: 'title and description required' });
    const blog = await Blog.create({ title, description, author: req.user._id });
    const populated = await blog.populate('author', 'username');
    res.status(201).json(populated);
  } catch (err) {
    res.status(500).json({ message: 'Error', error: err.message });
  }
});

// PUT /blogs/:id - update (only author)
router.put('/:id', auth, async (req, res) => {
  try {
    const { id } = req.params;
    const blog = await Blog.findById(id);
    if (!blog) return res.status(404).json({ message: 'Blog not found' });
    if (!blog.author.equals(req.user._id)) return res.status(403).json({ message: 'Not allowed' });

    const { title, description } = req.body;
    if (title) blog.title = title;
    if (description) blog.description = description;
    await blog.save();
    const populated = await blog.populate('author', 'username');
    res.json(populated);
  } catch (err) {
    res.status(500).json({ message: 'Error', error: err.message });
  }
});

// DELETE /blogs/:id - delete (only author)
router.delete('/:id', auth, async (req, res) => {
  try {
    const { id } = req.params;
    const blog = await Blog.findById(id);
    if (!blog) return res.status(404).json({ message: 'Blog not found' });
    if (!blog.author.equals(req.user._id)) return res.status(403).json({ message: 'Not allowed' });

    await blog.remove();
    res.json({ message: 'Blog deleted' });
  } catch (err) {
    res.status(500).json({ message: 'Error', error: err.message });
  }
```

```
});

module.exports = router;
```

Task 8 — Run Instructions & Example API Calls

Run instructions:

```
1. Fill .env (MONGO_URI, JWT_SECRET)
2. Start MongoDB locally or use Atlas
3. Start the server:
   npm run dev   # development with nodemon
   npm start     # production
4. Visit http://localhost:3000/ to see root route response.
```

Example API usage (curl):

```
# Signup
curl -X POST http://localhost:3000/auth/signup \
 -H "Content-Type: application/json" \
 -d '{"username":"saurav", "password":"pass123"}'

# Signin (get token)
curl -X POST http://localhost:3000/auth/signin \
 -H "Content-Type: application/json" \
 -d '{"username":"saurav", "password":"pass123"}'

# Create a blog (authenticated)
curl -X POST http://localhost:3000/blogs \
 -H "Authorization: Bearer <TOKEN>" \
 -H "Content-Type: application/json" \
 -d '{"title":"My first blog","description":"Hello world"}'

# Get all blogs
curl http://localhost:3000/blogs

# Update blog (only author)
curl -X PUT http://localhost:3000/blogs/<BLOG_ID> \
 -H "Authorization: Bearer <TOKEN>" \
 -H "Content-Type: application/json" \
 -d '{"title":"Updated title"}'

# Delete blog (only author)
curl -X DELETE http://localhost:3000/blogs/<BLOG_ID> \
 -H "Authorization: Bearer <TOKEN>"
```

Notes & Improvements

• For production, use strong JWT secret and consider refresh tokens.
• Add input validation (express-validator) to validate request bodies.
• Implement rate limiting and request logging (morgan/winston).
• Use HTTPS in production and store secrets securely (e.g., Vault).
• Add unit/integration tests and CI pipeline.
• Consider pagination for GET /blogs and search/filter options.

If you want, I can also: create a GitHub repo structure, provide a zipped project, or customize auth/roles.