# **Setting up the Backend**

This guide explains one way to organize a backend application and how each module works. We use a modular structure to separate configuration, routing, controllers, and middleware, which makes the code easier to understand, test, and maintain. Our authentication system leverages bcrypt for password hashing and JSON Web Tokens (JWT) for stateless session management.

## **Project Structure**

backend/

├── config/

│ └── database.js // Database connection using mysql2 and environment variables

├── routes/

│ └── auth.js // Routes for user registration and login

├── authMiddleware.js // Middleware to protect routes by verifying JWT tokens

├── app.js // Main Express application file that sets up middleware and routes

└── .env // Environment variables (DB creds, JWT secret, etc.)

### **What Each Part Does**

* **config/database.js:** Contains the MySQL connection pool setup using the mysql2 package. By using a pool, the application can efficiently manage multiple database connections. Sensitive details like host, user, and password are stored in the .env file.
* **routes/auth.js:** Defines our authentication routes. It includes endpoints for registration and login. During registration, passwords are hashed using bcrypt. During login, the password is compared to the stored hash, and if valid, a JWT is created and sent to the client.
* **authMiddleware.js:** Provides a middleware function that checks for a valid JWT in the request cookies. Protected routes can use this middleware to ensure that only authenticated users have access.
* **app.js:** The entry point of the application. It initializes Express, loads middleware (for JSON body parsing and cookie parsing), and mounts the authentication routes.
* **.env:** A file containing configuration variables such as database credentials and the secret used to sign JWTs. This file should never be committed to version control.

## **Environment Variables**

Create a .env file in the backend folder with content similar to:

DB\_HOST=localhost

DB\_USER=root

DB\_PASSWORD=password

DB\_NAME=authdb

DB\_PORT=3306

PORT=3000

JWT\_SECRET=your\_jwt\_secret\_here

**Explanation:**

* The database connection details and server port are stored here.
* **JWT\_SECRET** is used to sign and verify JWTs. Use a strong, unpredictable secret.

## **Configuring the Database Connection**

File: backend/config/database.js

const mysql = require('mysql2');

require('dotenv').config();

const pool = mysql.createPool({

host: process.env.DB\_HOST,

user: process.env.DB\_USER,

password: process.env.DB\_PASSWORD,

database: process.env.DB\_NAME,

port: process.env.DB\_PORT || 3306,

waitForConnections: true,

connectionLimit: 10,

queueLimit: 0

});

module.exports = pool;

**Explanation:**

* We use the mysql2 library’s connection pool for efficient database interactions.
* The pool reads configuration from the .env file, keeping sensitive data out of the codebase.

## **Authentication Routes**

File: backend/routes/auth.js

const express = require('express');

const bcrypt = require('bcryptjs');

const jwt = require('jsonwebtoken');

const pool = require('../config/database'); // Note: Using ../config since auth.js is in the routes folder

const router = express.Router();

/\*\*

\* Registers a new user.

\* Hashes the password using bcrypt and stores the user in the database.

\*/

router.post('/register', async (req, res) => {

const { username, password } = req.body;

try {

// Hash the user's password with a salt round of 10

const hashedPassword = await bcrypt.hash(password, 10);

// Insert the user into the database using parameterized queries to avoid SQL injection

const [result] = await pool.promise().execute(

'INSERT INTO users (username, password) VALUES (?, ?)',

[username, hashedPassword]

);

res.status(201).json({ message: 'User registered successfully' });

} catch (error) {

console.error("Error during user registration:", error);

res.status(500).json({ error: 'Failed to register user' });

}

});

/\*\*

\* Logs in a user.

\* Verifies credentials, generates a JWT if valid, and sends it in an HTTP-only cookie.

\*/

router.post('/login', async (req, res) => {

const { username, password } = req.body;

try {

// Query the database for the user by username

const [rows] = await pool.promise().execute(

'SELECT \* FROM users WHERE username = ?',

[username]

);

const user = rows[0];

// Check if the user exists and if the password is correct

if (user && await bcrypt.compare(password, user.password)) {

// Create a JWT payload containing user details

const token = jwt.sign(

{ id: user.id, username: user.username },

process.env.JWT\_SECRET,

{ expiresIn: '1h' }

);

// Set the token in an HTTP-only cookie to protect it from client-side scripts

res.cookie('token', token, { httpOnly: true, sameSite: 'strict' });

res.json({ message: 'Login successful' });

} else {

res.status(401).json({ error: 'Invalid credentials' });

}

} catch (error) {

console.error("Error during login:", error);

res.status(500).json({ error: 'Failed to login' });

}

});

module.exports = router;

**Detailed Explanations:**

* **Registration (/register):**
  + **bcrypt.hash:** The plaintext password is hashed using bcrypt with 10 salt rounds. This ensures that even if the database is compromised, attackers cannot easily reverse the hash to obtain the original password.
  + **Parameterized Query:** The query uses placeholders (?) to safely insert values, protecting against SQL injection.
* **Login (/login):**
  + **bcrypt.compare:** Compares the provided plaintext password with the stored hashed password.
  + **JWT Generation:** If authentication is successful, we generate a JWT using jwt.sign. The token includes user data (like id and username) and is set to expire in 1 hour.
  + **HTTP-only Cookie:** The JWT is stored in a cookie with the httpOnly flag, which prevents client-side scripts from accessing it, adding an extra layer of security.

## **JWT and Security Techniques**

* **JSON Web Tokens (JWT):** JWTs allow for stateless authentication. Once a user is authenticated, the server issues a token that the client sends with each request. This token proves the user's identity without needing to query the database repeatedly.
* **Bcrypt:** Bcrypt is a password hashing function that adds salt (random data) to the password before hashing. This prevents attackers from using precomputed tables (rainbow tables) to reverse the hash.
* **HTTP-only Cookies:** Storing JWTs in HTTP-only cookies prevents client-side scripts from accessing the token, which helps mitigate XSS (Cross-Site Scripting) attacks.

## **Auth Middleware**

File: backend/authMiddlewrae.js

const jwt = require('jsonwebtoken');

/\*\*

\* Middleware to protect routes by verifying JWT in cookies.

\*/

const authMiddleware = (req, res, next) => {

// Retrieve the token from cookies

const token = req.cookies.token;

if (!token) return res.status(403).json({ error: 'Access denied' });

try {

// Verify the token using the secret

const decoded = jwt.verify(token, process.env.JWT\_SECRET);

req.user = decoded; // Attach the decoded token (user info) to the request

next(); // Proceed to the next middleware or route handler

} catch (error) {

res.status(401).json({ error: 'Invalid token' });

}

};

module.exports = authMiddleware;

**Explanation:**

* The middleware checks if a JWT is present in the request cookies.
* It verifies the token using the secret defined in the environment variables.
* If the token is valid, the middleware attaches the decoded user information to the request object and calls next(), allowing access to protected routes.
* If verification fails, it sends an error response.

## **Setting Up the Express Application**

File: backend/app.js

const express = require('express');

const cookieParser = require('cookie-parser'); // Needed to parse cookies from requests

require('dotenv').config();

const app = express();

// Middleware for parsing JSON bodies and cookies

app.use(express.json());

app.use(cookieParser());

// Mount the authentication routes under /auth

const authRoutes = require('./routes/auth'); // Adjust the path if necessary

app.use('/auth', authRoutes);

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

app.listen(PORT, () => {

console.log(`Server running on port ${PORT}`);

});

**Explanation:**

* **Middleware:**
  + express.json() parses incoming JSON payloads.
  + cookieParser() enables the server to read cookies from incoming requests.
* **Mounting Routes:** The authentication routes are mounted under the /auth path. This means registration is accessible via /auth/register and login via /auth/login.
* **Starting the Server:** The server listens on the port specified in the .env file (default 3000).

## **Final Remarks**

This guide demonstrates a modular approach to building a secure Node.js backend:

* **Modularity:** Separating concerns into configuration, routes, and middleware makes the application easier to maintain and extend.
* **Security:** Using bcrypt for password hashing and JWTs for stateless authentication enhances security. HTTP-only cookies help protect tokens from client-side vulnerabilities.
* **Scalability:** The structure is designed to be scalable. As your application grows, you can add more routes, controllers, and middleware without cluttering the codebase.

By following this guide, beginners can understand how each component works and how they interact to provide a secure authentication mechanism. This modular structure also makes it easier to transition to a production environment, such as Azure, with minimal changes—primarily by updating the environment variables.

Happy coding!