import express from "express";

import helmet from "helmet";

import rateLimit from "express-rate-limit";

import csurf from "csurf";

import cookieParser from "cookie-parser";

import { body, validationResult } from "express-validator";

import bcrypt from "bcrypt";

import jwt from "jsonwebtoken";

import sanitizeHtml from "sanitize-html";

import { Pool } from "pg"; // use parameterized queries

const app = express();

app.use(helmet()); // secure headers

app.use(express.json({ limit: "10kb" })); // limit size

app.use(cookieParser());

const db = new Pool({ connectionString: process.env.DATABASE\_URL });

// Basic rate limiter (tune as needed)

const limiter = rateLimit({

windowMs: 60\_000, // 1 minute

max: 60, // 60 requests per IP per window

});

app.use(limiter);

// CSRF protection for state-changing endpoints (uses cookies)

const csrfProtection = csurf({

cookie: { httpOnly: true, sameSite: "lax", secure: process.env.NODE\_ENV === "production" },

});

// secure cookie helper

function setAuthCookie(res, token) {

res.cookie("auth", token, {

httpOnly: true,

secure: process.env.NODE\_ENV === "production",

sameSite: "lax",

maxAge: 1000 \* 60 \* 60 \* 24, // 1 day

});

}

// --- Auth endpoints (signup / login) ---

// Signup

app.post(

"/api/signup",

[

body("email").isEmail().normalizeEmail(),

body("password").isLength({ min: 10 }),

body("displayName").trim().isLength({ min: 1, max: 50 }).escape(),

],

async (req, res) => {

const errors = validationResult(req);

if (!errors.isEmpty()) return res.status(400).json({ errors: errors.array() });

const { email, password, displayName } = req.body;

const hashed = await bcrypt.hash(password, 12);

// parameterized query prevents SQL injection

await db.query("INSERT INTO users(email, password\_hash, display\_name) VALUES($1,$2,$3)", [

email,

hashed,

displayName,

]);

// Create email verification token, send email (not shown)

res.status(201).json({ message: "Account created. Verify your email." });

}

);

// Login (returns JWT in secure cookie)

app.post(

"/api/login",

[body("email").isEmail().normalizeEmail(), body("password").exists()],

async (req, res) => {

const { email, password } = req.body;

const { rows } = await db.query("SELECT id, password\_hash FROM users WHERE email = $1", [email]);

if (!rows[0]) return res.status(401).json({ error: "Invalid credentials" });

const ok = await bcrypt.compare(password, rows[0].password\_hash);

if (!ok) return res.status(401).json({ error: "Invalid credentials" });

const token = jwt.sign({ sub: rows[0].id }, process.env.JWT\_SECRET, { expiresIn: "1d" });

setAuthCookie(res, token);

res.json({ message: "Logged in" });

}

);

// Auth middleware (verifies JWT)

function authMiddleware(req, res, next) {

const token = req.cookies.auth;

if (!token) return res.status(401).json({ error: "Unauthorized" });

try {

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

req.userId = payload.sub;

next();

} catch (e) {

return res.status(401).json({ error: "Unauthorized" });

}

}

// --- Posts endpoints ---

// Create post (authenticated) with CSRF protection

app.post(

"/api/posts",

authMiddleware,

csrfProtection,

[

body("title").trim().isLength({ min: 1, max: 200 }),

body("content").isString().isLength({ min: 1, max: 20000 }),

],

async (req, res) => {

const errors = validationResult(req);

if (!errors.isEmpty()) return res.status(400).json({ errors: errors.array() });

// Sanitize content to remove harmful scripts but allow some formatting

const cleanContent = sanitizeHtml(req.body.content, {

allowedTags: ["b", "i", "em", "strong", "a", "p", "ul", "ol", "li", "br", "blockquote", "code"],

allowedAttributes: { a: ["href", "rel", "target"] },

transformTags: {

a: (tagName, attribs) => {

// force rel and target to prevent target=\_blank attacks

return { tagName: "a", attribs: { href: attribs.href, rel: "nofollow noopener", target: "\_blank" } };

},

},

});

const { title } = req.body;

const result = await db.query(

"INSERT INTO posts(author\_id, title, content, created\_at) VALUES($1,$2,$3,now()) RETURNING id",

[req.userId, title, cleanContent]

);

res.status(201).json({ id: result.rows[0].id });

}

);

// Update post: ownership check (authorization)

app.put(

"/api/posts/:id",

authMiddleware,

csrfProtection,

[body("title").optional().trim().isLength({ min: 1, max: 200 }), body("content").optional().isString()],

async (req, res) => {

const errors = validationResult(req);

if (!errors.isEmpty()) return res.status(400).json({ errors: errors.array() });

const postId = Number(req.params.id);

// ownership check

const post = await db.query("SELECT author\_id FROM posts WHERE id = $1", [postId]);

if (!post.rows.length) return res.status(404).json({ error: "Not found" });

if (post.rows[0].author\_id !== req.userId) return res.status(403).json({ error: "Forbidden" });

const updates = [];

const params = [];

let idx = 1;

if (req.body.title) {

updates.push(`title = $${idx++}`);

params.push(req.body.title);

}

if (req.body.content) {

updates.push(`content = $${idx++}`);

params.push(sanitizeHtml(req.body.content, { allowedTags: ["p", "b", "i", "a"] }));

}

if (updates.length === 0) return res.status(400).json({ error: "Nothing to update" });

params.push(postId);

await db.query(`UPDATE posts SET ${updates.join(", ")} WHERE id = $${idx}`, params);

res.json({ message: "Updated" });

}

);

// Error handler (no stack traces in production)

app.use((err, req, res, next) => {

console.error(err);

if (process.env.NODE\_ENV === "production") return res.status(500).json({ error: "Internal Server Error" });

res.status(500).json({ error: err.message, stack: err.stack });

});

export default app;