INTERNAL PRACTICAL EXAM

NECESSARY CODES:

db.ts:

import { drizzle } from "drizzle-orm/postgres-js";

import postgres from "postgres";

import \* as schema from "@shared/schema";

// Create PostgreSQL connection

const connectionString = process.env.DATABASE\_URL;

if (!connectionString) {

  throw new Error("DATABASE\_URL environment variable is not set");

}

// For use with drizzle

const client = postgres(connectionString, { max: 1 });

// Create Drizzle instance

export const db = drizzle(client, { schema });

index.ts:

import './models/mongodb'; // Initialize MongoDB connection

import express, { type Request, Response, NextFunction } from "express";

import { registerRoutes } from "./routes";

import { setupVite, serveStatic, log } from "./vite";

const app = express();

app.use(express.json());

app.use(express.urlencoded({ extended: false }));

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

  const start = Date.now();

  const path = req.path;

  let capturedJsonResponse: Record<string, any> | undefined = undefined;

  const originalResJson = res.json;

  res.json = function (bodyJson, ...args) {

    capturedJsonResponse = bodyJson;

    return originalResJson.apply(res, [bodyJson, ...args]);

  };

  res.on("finish", () => {

    const duration = Date.now() - start;

    if (path.startsWith("/api")) {

      let logLine = `${req.method} ${path} ${res.statusCode} in ${duration}ms`;

      if (capturedJsonResponse) {

        logLine += ` :: ${JSON.stringify(capturedJsonResponse)}`;

      }

      if (logLine.length > 80) {

        logLine = logLine.slice(0, 79) + "…";

      }

      log(logLine);

    }

  });

  next();

});

(async () => {

  const server = await registerRoutes(app);

  app.use((err: any, \_req: Request, res: Response, \_next: NextFunction) => {

    const status = err.status || err.statusCode || 500;

    const message = err.message || "Internal Server Error";

    res.status(status).json({ message });

    throw err;

  });

  // importantly only setup vite in development and after

  // setting up all the other routes so the catch-all route

  // doesn't interfere with the other routes

  if (app.get("env") === "development") {

    await setupVite(app, server);

  } else {

    serveStatic(app);

  }

  // ALWAYS serve the app on port 5000

  // this serves both the API and the client.

  // It is the only port that is not firewalled.

  const port = 5000;

  server.listen({

    port,

    host: "0.0.0.0",

    reusePort: true,

  }, () => {

    log(`serving on port ${port}`);

  });

})();

Routes.ts:

import type { Express, Request, Response } from "express";

import { createServer, type Server } from "http";

import { mongoStorage } from "./mongo-storage";

import multer from "multer";

import sharp from "sharp";

import path from "path";

import fs from "fs";

import { insertCompressionLogSchema } from "./models/mongodb";

import { z } from "zod";

// Set up multer for file uploads

const upload = multer({

  storage: multer.memoryStorage(),

  limits: { fileSize: 10 \* 1024 \* 1024 }, // 10MB limit

  fileFilter: (\_req, file, cb) => {

    // Accept only image files

    const filetypes = /jpeg|jpg|png|gif|webp/;

    const mimetype = filetypes.test(file.mimetype);

    const extname = filetypes.test(path.extname(file.originalname).toLowerCase());

    if (mimetype && extname) {

      return cb(null, true);

    }

    cb(new Error("Error: Images Only!"));

  },

});

// Create a temporary directory for storing processed images

const tempDir = path.join(process.cwd(), "temp");

if (!fs.existsSync(tempDir)) {

  fs.mkdirSync(tempDir, { recursive: true });

}

export async function registerRoutes(app: Express): Promise<Server> {

  // put application routes here

  // prefix all routes with /api

  // Get analytics data

  app.get("/api/analytics", async (\_req: Request, res: Response) => {

    try {

      const stats = await mongoStorage.getCompressionStats();

      const dailyStats = await mongoStorage.getDailyCompressionStats(7);

      res.json({

        ...stats,

        dailyStats

      });

    } catch (error) {

      console.error("Error fetching analytics:", error);

      res.status(500).json({ message: "Failed to fetch analytics data" });

    }

  });

  // Get recent activity

  app.get("/api/activity", async (req: Request, res: Response) => {

    try {

      const limit = req.query.limit ? parseInt(req.query.limit as string) : 10;

      const logs = await mongoStorage.getCompressionLogs(limit);

      res.json(logs);

    } catch (error) {

      console.error("Error fetching activity logs:", error);

      res.status(500).json({ message: "Failed to fetch activity logs" });

    }

  });

  // Upload and compress image

  app.post("/api/compress", upload.single("image"), async (req: Request, res: Response) => {

    try {

      if (!req.file) {

        return res.status(400).json({ message: "No image file uploaded" });

      }

      const compressionOptions = {

        quality: parseInt(req.body.quality || "75"),

        format: req.body.format || "jpeg",

        width: req.body.width ? parseInt(req.body.width) : undefined,

        height: req.body.height ? parseInt(req.body.height) : undefined,

        maintainAspectRatio: req.body.maintainAspectRatio === "true"

      };

      // Get image info before compression

      const originalImageInfo = await sharp(req.file.buffer).metadata();

      const originalSize = req.file.size;

      // Create Sharp instance

      let sharpInstance = sharp(req.file.buffer);

      // Resize if needed

      if (compressionOptions.width || compressionOptions.height) {

        sharpInstance = sharpInstance.resize({

          width: compressionOptions.width,

          height: compressionOptions.height,

          fit: compressionOptions.maintainAspectRatio ? "inside" : "fill"

        });

      }

      // Set format and quality

      switch (compressionOptions.format) {

        case "jpeg":

        case "jpg":

          sharpInstance = sharpInstance.jpeg({ quality: compressionOptions.quality });

          break;

        case "png":

          sharpInstance = sharpInstance.png({ quality: compressionOptions.quality });

          break;

        case "webp":

          sharpInstance = sharpInstance.webp({ quality: compressionOptions.quality });

          break;

        case "avif":

          sharpInstance = sharpInstance.avif({ quality: compressionOptions.quality });

          break;

        default:

          sharpInstance = sharpInstance.jpeg({ quality: compressionOptions.quality });

      }

      // Process the image

      const compressedBuffer = await sharpInstance.toBuffer();

      const compressedInfo = await sharp(compressedBuffer).metadata();

      const compressedSize = compressedBuffer.length;

      // Calculate compression ratio

      const compressionRatio = Math.round((1 - (compressedSize / originalSize)) \* 100);

      // Generate a unique filename

      const timestamp = Date.now();

      const filename = `${path.parse(req.file.originalname).name}-compressed-${timestamp}.${compressionOptions.format}`;

      const outputPath = path.join(tempDir, filename);

      // Save the compressed image temporarily

      await fs.promises.writeFile(outputPath, compressedBuffer);

      // Log the compression

      const logData = {

        filename: req.file.originalname,

        originalSize,

        compressedSize,

        compressionRatio,

        format: compressionOptions.format,

        width: compressedInfo.width || 0,

        height: compressedInfo.height || 0,

        quality: compressionOptions.quality,

        metadata: {

          originalWidth: originalImageInfo.width,

          originalHeight: originalImageInfo.height,

          resize: !!(compressionOptions.width || compressionOptions.height)

        }

      };

      // Validate the log data

      const validatedData = insertCompressionLogSchema.parse(logData);

      const compressionLog = await mongoStorage.createCompressionLog(validatedData);

      res.json({

        success: true,

        original: {

          size: originalSize,

          width: originalImageInfo.width,

          height: originalImageInfo.height,

          filename: req.file.originalname

        },

        compressed: {

          size: compressedSize,

          width: compressedInfo.width,

          height: compressedInfo.height,

          filename,

          compressionRatio,

          downloadUrl: `/api/download/${filename}`

        },

        log: compressionLog

      });

    } catch (error) {

      console.error("Error compressing image:", error);

      if (error instanceof z.ZodError) {

        return res.status(400).json({ message: "Invalid compression data", errors: error.errors });

      }

      res.status(500).json({ message: "Failed to compress image" });

    }

  });

  // Download compressed image

  app.get("/api/download/:filename", (req: Request, res: Response) => {

    try {

      const filename = req.params.filename;

      const filePath = path.join(tempDir, filename);

      if (!fs.existsSync(filePath)) {

        return res.status(404).json({ message: "File not found" });

      }

      res.download(filePath);

    } catch (error) {

      console.error("Error downloading file:", error);

      res.status(500).json({ message: "Failed to download file" });

    }

  });

  // Clear temporary files that are older than 1 hour

  setInterval(() => {

    try {

      const files = fs.readdirSync(tempDir);

      const now = Date.now();

      files.forEach(file => {

        const filePath = path.join(tempDir, file);

        const stats = fs.statSync(filePath);

        const fileAge = now - stats.mtimeMs;

        // Delete files older than 1 hour (3600000 ms)

        if (fileAge > 3600000) {

          fs.unlinkSync(filePath);

        }

      });

    } catch (error) {

      console.error("Error cleaning up temp directory:", error);

    }

  }, 3600000); // Run every hour

  const httpServer = createServer(app);

  return httpServer;

}

Storage.ts:

import { users, compressionLogs, type User, type InsertUser, type CompressionLog, type InsertCompressionLog } from "@shared/schema";

import { db } from "./db";

import { eq, desc, sql, gte } from "drizzle-orm";

import { getLocalISOString } from "../shared/utils";

export interface IStorage {

  getUser(id: number): Promise<User | undefined>;

  getUserByUsername(username: string): Promise<User | undefined>;

  createUser(user: InsertUser): Promise<User>;

  // Compression logs

  createCompressionLog(log: InsertCompressionLog): Promise<CompressionLog>;

  getCompressionLogs(limit?: number): Promise<CompressionLog[]>;

  getCompressionLogsByUserId(userId?: number, limit?: number): Promise<CompressionLog[]>;

  getCompressionStats(): Promise<{

    totalCompressions: number;

    totalSizeSaved: number;

    averageCompressionRatio: number;

    averageSizeReduced: number;

  }>;

  getDailyCompressionStats(days: number): Promise<{

    date: string;

    count: number;

  }[]>;

}

export class DatabaseStorage implements IStorage {

  async getUser(id: number): Promise<User | undefined> {

    const [user] = await db.select().from(users).where(eq(users.id, id));

    return user;

  }

  async getUserByUsername(username: string): Promise<User | undefined> {

    const [user] = await db.select().from(users).where(eq(users.username, username));

    return user;

  }

  async createUser(insertUser: InsertUser): Promise<User> {

    const [user] = await db

      .insert(users)

      .values(insertUser)

      .returning();

    return user;

  }

  async createCompressionLog(insertLog: InsertCompressionLog): Promise<CompressionLog> {

    const [log] = await db

      .insert(compressionLogs)

      .values(insertLog)

      .returning();

    return log;

  }

  async getCompressionLogs(limit = 10): Promise<CompressionLog[]> {

    return await db

      .select()

      .from(compressionLogs)

      .orderBy(desc(compressionLogs.createdAt))

      .limit(limit);

  }

  async getCompressionLogsByUserId(userId?: number, limit = 10): Promise<CompressionLog[]> {

    if (userId) {

      return await db

        .select()

        .from(compressionLogs)

        .where(eq(compressionLogs.userId, userId))

        .orderBy(desc(compressionLogs.createdAt))

        .limit(limit);

    } else {

      return await this.getCompressionLogs(limit);

    }

  }

  async getCompressionStats(): Promise<{

    totalCompressions: number;

    totalSizeSaved: number;

    averageCompressionRatio: number;

    averageSizeReduced: number;

  }> {

    const [result] = await db

      .select({

        totalCompressions: sql<number>`cast(count(\*) as integer)`,

        totalSizeSaved: sql<number>`cast(coalesce(sum(${compressionLogs.originalSize} - ${compressionLogs.compressedSize}), 0) as integer)`,

        averageCompressionRatio: sql<number>`cast(coalesce(avg(${compressionLogs.compressionRatio}), 0) as integer)`,

        averageSizeReduced: sql<number>`cast(coalesce(avg(${compressionLogs.originalSize} - ${compressionLogs.compressedSize}), 0) as integer)`

      })

      .from(compressionLogs);

    return {

      totalCompressions: result.totalCompressions || 0,

      totalSizeSaved: result.totalSizeSaved || 0,

      averageCompressionRatio: result.averageCompressionRatio || 0,

      averageSizeReduced: result.averageSizeReduced || 0

    };

  }

  async getDailyCompressionStats(days = 7): Promise<{ date: string; count: number }[]> {

    // Get the date 'days' days ago

    const startDate = new Date();

    startDate.setDate(startDate.getDate() - (days - 1));

    startDate.setHours(0, 0, 0, 0);

    // Query to get the daily count

    const results = await db

      .select({

        date: sql<string>`to\_char(${compressionLogs.createdAt}, 'YYYY-MM-DD')`,

        count: sql<number>`cast(count(\*) as integer)`

      })

      .from(compressionLogs)

      .where(gte(compressionLogs.createdAt, startDate))

      .groupBy(sql`to\_char(${compressionLogs.createdAt}, 'YYYY-MM-DD')`)

      .orderBy(sql`to\_char(${compressionLogs.createdAt}, 'YYYY-MM-DD')`);

    // Create a map of date to count from the results

    const dateCountMap = new Map();

    results.forEach(result => {

      dateCountMap.set(result.date, result.count);

    });

    // Generate all dates in the range with zero counts for missing dates

    const stats = [];

    for (let i = 0; i < days; i++) {

      const date = new Date(startDate);

      date.setDate(date.getDate() + i);

      const dateString = getLocalISOString(date).split('T')[0];

      stats.push({

        date: dateString,

        count: dateCountMap.get(dateString) || 0

      });

    }

    return stats;

  }

}

// Using the database implementation

export const storage = new DatabaseStorage();

vite.ts:

import express, { type Express } from "express";

import fs from "fs";

import path from "path";

import { createServer as createViteServer, createLogger } from "vite";

import { type Server } from "http";

import viteConfig from "../vite.config";

import { nanoid } from "nanoid";

const viteLogger = createLogger();

export function log(message: string, source = "express") {

  const formattedTime = new Date().toLocaleTimeString("en-US", {

    hour: "numeric",

    minute: "2-digit",

    second: "2-digit",

    hour12: true,

  });

  console.log(`${formattedTime} [${source}] ${message}`);

}

export async function setupVite(app: Express, server: Server) {

  const serverOptions = {

    middlewareMode: true,

    hmr: { server },

    allowedHosts: true,

  };

  const vite = await createViteServer({

    ...viteConfig,

    configFile: false,

    customLogger: {

      ...viteLogger,

      error: (msg, options) => {

        viteLogger.error(msg, options);

        process.exit(1);

      },

    },

    server: serverOptions,

    appType: "custom",

  });

  app.use(vite.middlewares);

  app.use("\*", async (req, res, next) => {

    const url = req.originalUrl;

    try {

      const clientTemplate = path.resolve(

        import.meta.dirname,

        "..",

        "client",

        "index.html",

      );

      // always reload the index.html file from disk incase it changes

      let template = await fs.promises.readFile(clientTemplate, "utf-8");

      template = template.replace(

        `src="/src/main.tsx"`,

        `src="/src/main.tsx?v=${nanoid()}"`,

      );

      const page = await vite.transformIndexHtml(url, template);

      res.status(200).set({ "Content-Type": "text/html" }).end(page);

    } catch (e) {

      vite.ssrFixStacktrace(e as Error);

      next(e);

    }

  });

}

export function serveStatic(app: Express) {

  const distPath = path.resolve(import.meta.dirname, "public");

  if (!fs.existsSync(distPath)) {

    throw new Error(

      `Could not find the build directory: ${distPath}, make sure to build the client first`,

    );

  }

  app.use(express.static(distPath));

  // fall through to index.html if the file doesn't exist

  app.use("\*", (\_req, res) => {

    res.sendFile(path.resolve(distPath, "index.html"));

  });

}

Schema.ts:

import { pgTable, text, serial, integer, boolean, timestamp, json } from "drizzle-orm/pg-core";

import { createInsertSchema } from "drizzle-zod";

import { z } from "zod";

export const users = pgTable("users", {

  id: serial("id").primaryKey(),

  username: text("username").notNull().unique(),

  password: text("password").notNull(),

});

export const compressionLogs = pgTable("compression\_logs", {

  id: serial("id").primaryKey(),

  filename: text("filename").notNull(),

  originalSize: integer("original\_size").notNull(), // in bytes

  compressedSize: integer("compressed\_size").notNull(), // in bytes

  compressionRatio: integer("compression\_ratio").notNull(), // percentage

  format: text("format").notNull(), // jpg, png, webp, etc.

  width: integer("width").notNull(),

  height: integer("height").notNull(),

  quality: integer("quality").notNull(), // 1-100

  createdAt: timestamp("created\_at").defaultNow().notNull(),

  userId: integer("user\_id").references(() => users.id, { onDelete: "cascade" }),

  metadata: json("metadata").$type<{

    originalWidth?: number;

    originalHeight?: number;

    resize?: boolean;

  }>(),

});

export const insertUserSchema = createInsertSchema(users).pick({

  username: true,

  password: true,

});

export const insertCompressionLogSchema = createInsertSchema(compressionLogs).omit({

  id: true,

  createdAt: true,

});

export type InsertUser = z.infer<typeof insertUserSchema>;

export type User = typeof users.$inferSelect;

export type CompressionLog = typeof compressionLogs.$inferSelect;

export type InsertCompressionLog = z.infer<typeof insertCompressionLogSchema>;

utils.ts:

/\*\*

 \* Get an ISO string that preserves the local timezone information

 \* @param date Date object to convert

 \* @returns ISO string with local timezone

 \*/

export function getLocalISOString(date: Date): string {

  const offset = date.getTimezoneOffset();

  const localDate = new Date(date.getTime() - offset \* 60 \* 1000);

  return localDate.toISOString();

}

{

  "name": "rest-express",

  "version": "1.0.0",

  "type": "module",

  "license": "MIT",

  "scripts": {

    "dev": "tsx server/index.ts",

    "build": "vite build && esbuild server/index.ts --platform=node --packages=external --bundle --format=esm --outdir=dist",

    "start": "NODE\_ENV=production node dist/index.js",

    "check": "tsc",

    "db:push": "drizzle-kit push"

  },

  "dependencies": {

    "@hookform/resolvers": "^3.9.1",

    "@jridgewell/trace-mapping": "^0.3.25",

    "@neondatabase/serverless": "^0.10.4",

    "@radix-ui/react-accordion": "^1.2.1",

    "@radix-ui/react-alert-dialog": "^1.1.2",

    "@radix-ui/react-aspect-ratio": "^1.1.0",

    "@radix-ui/react-avatar": "^1.1.1",

    "@radix-ui/react-checkbox": "^1.1.2",

    "@radix-ui/react-collapsible": "^1.1.1",

    "@radix-ui/react-context-menu": "^2.2.2",

    "@radix-ui/react-dialog": "^1.1.2",

    "@radix-ui/react-dropdown-menu": "^2.1.2",

    "@radix-ui/react-hover-card": "^1.1.2",

    "@radix-ui/react-label": "^2.1.0",

    "@radix-ui/react-menubar": "^1.1.2",

    "@radix-ui/react-navigation-menu": "^1.2.1",

    "@radix-ui/react-popover": "^1.1.2",

    "@radix-ui/react-progress": "^1.1.0",

    "@radix-ui/react-radio-group": "^1.2.1",

    "@radix-ui/react-scroll-area": "^1.2.0",

    "@radix-ui/react-select": "^2.1.2",

    "@radix-ui/react-separator": "^1.1.0",

    "@radix-ui/react-slider": "^1.2.1",

    "@radix-ui/react-slot": "^1.1.0",

    "@radix-ui/react-switch": "^1.1.1",

    "@radix-ui/react-tabs": "^1.1.1",

    "@radix-ui/react-toast": "^1.2.2",

    "@radix-ui/react-toggle": "^1.1.0",

    "@radix-ui/react-toggle-group": "^1.1.0",

    "@radix-ui/react-tooltip": "^1.1.3",

    "@replit/vite-plugin-shadcn-theme-json": "^0.0.4",

    "@tanstack/react-query": "^5.60.5",

    "class-variance-authority": "^0.7.0",

    "clsx": "^2.1.1",

    "cmdk": "^1.0.0",

    "connect-pg-simple": "^10.0.0",

    "date-fns": "^3.6.0",

    "drizzle-orm": "^0.39.1",

    "drizzle-zod": "^0.7.0",

    "embla-carousel-react": "^8.3.0",

    "express": "^4.21.2",

    "express-session": "^1.18.1",

    "framer-motion": "^11.13.1",

    "input-otp": "^1.2.4",

    "lucide-react": "^0.453.0",

    "memorystore": "^1.6.7",

    "mongodb": "^6.15.0",

    "mongoose": "^8.13.2",

    "multer": "^1.4.5-lts.2",

    "passport": "^0.7.0",

    "passport-local": "^1.0.0",

    "postgres": "^3.4.5",

    "react": "^18.3.1",

    "react-day-picker": "^8.10.1",

    "react-dom": "^18.3.1",

    "react-hook-form": "^7.53.1",

    "react-icons": "^5.4.0",

    "react-resizable-panels": "^2.1.4",

    "recharts": "^2.13.0",

    "sharp": "^0.33.5",

    "tailwind-merge": "^2.5.4",

    "tailwindcss-animate": "^1.0.7",

    "vaul": "^1.1.0",

    "wouter": "^3.3.5",

    "ws": "^8.18.0",

    "zod": "^3.23.8",

    "zod-validation-error": "^3.4.0"

  },

  "devDependencies": {

    "@replit/vite-plugin-cartographer": "^0.0.11",

    "@replit/vite-plugin-runtime-error-modal": "^0.0.3",

    "@tailwindcss/typography": "^0.5.15",

    "@types/connect-pg-simple": "^7.0.3",

    "@types/express": "4.17.21",

    "@types/express-session": "^1.18.0",

    "@types/node": "20.16.11",

    "@types/passport": "^1.0.16",

    "@types/passport-local": "^1.0.38",

    "@types/react": "^18.3.11",

    "@types/react-dom": "^18.3.1",

    "@types/ws": "^8.5.13",

    "@vitejs/plugin-react": "^4.3.2",

    "autoprefixer": "^10.4.20",

    "drizzle-kit": "^0.19.1",

    "esbuild": "^0.25.0",

    "postcss": "^8.4.47",

    "tailwindcss": "^3.4.14",

    "tsx": "^4.19.1",

    "typescript": "5.6.3",

    "vite": "^5.4.14"

  },

  "optionalDependencies": {

    "bufferutil": "^4.0.8"

  }

}

Tailwind.config.ts:

import type { Config } from "tailwindcss";

export default {

  darkMode: ["class"],

  content: ["./client/index.html", "./client/src/\*\*/\*.{js,jsx,ts,tsx}"],

  theme: {

    extend: {

      borderRadius: {

        lg: "var(--radius)",

        md: "calc(var(--radius) - 2px)",

        sm: "calc(var(--radius) - 4px)",

      },

      colors: {

        background: "hsl(var(--background))",

        foreground: "hsl(var(--foreground))",

        card: {

          DEFAULT: "hsl(var(--card))",

          foreground: "hsl(var(--card-foreground))",

        },

        popover: {

          DEFAULT: "hsl(var(--popover))",

          foreground: "hsl(var(--popover-foreground))",

        },

        primary: {

          DEFAULT: "hsl(var(--primary))",

          foreground: "hsl(var(--primary-foreground))",

        },

        secondary: {

          DEFAULT: "hsl(var(--secondary))",

          foreground: "hsl(var(--secondary-foreground))",

        },

        muted: {

          DEFAULT: "hsl(var(--muted))",

          foreground: "hsl(var(--muted-foreground))",

        },

        accent: {

          DEFAULT: "hsl(var(--accent))",

          foreground: "hsl(var(--accent-foreground))",

        },

        destructive: {

          DEFAULT: "hsl(var(--destructive))",

          foreground: "hsl(var(--destructive-foreground))",

        },

        border: "hsl(var(--border))",

        input: "hsl(var(--input))",

        ring: "hsl(var(--ring))",

        chart: {

          "1": "hsl(var(--chart-1))",

          "2": "hsl(var(--chart-2))",

          "3": "hsl(var(--chart-3))",

          "4": "hsl(var(--chart-4))",

          "5": "hsl(var(--chart-5))",

        },

        sidebar: {

          DEFAULT: "hsl(var(--sidebar-background))",

          foreground: "hsl(var(--sidebar-foreground))",

          primary: "hsl(var(--sidebar-primary))",

          "primary-foreground": "hsl(var(--sidebar-primary-foreground))",

          accent: "hsl(var(--sidebar-accent))",

          "accent-foreground": "hsl(var(--sidebar-accent-foreground))",

          border: "hsl(var(--sidebar-border))",

          ring: "hsl(var(--sidebar-ring))",

        },

      },

      keyframes: {

        "accordion-down": {

          from: {

            height: "0",

          },

          to: {

            height: "var(--radix-accordion-content-height)",

          },

        },

        "accordion-up": {

          from: {

            height: "var(--radix-accordion-content-height)",

          },

          to: {

            height: "0",

          },

        },

      },

      animation: {

        "accordion-down": "accordion-down 0.2s ease-out",

        "accordion-up": "accordion-up 0.2s ease-out",

      },

    },

  },

  plugins: [require("tailwindcss-animate"), require("@tailwindcss/typography")],

} satisfies Config;