# VIP Elite K9s Complete System Export

This is the complete codebase for the VIP Elite K9s staff and client management system.

## Package.json

```json

{

"name": "vip-elite-k9s",

"private": true,

"version": "0.0.0",

"type": "module",

"scripts": {

"dev": "tsx server/index.ts",

"build": "npm run build:frontend && npm run build:backend",

"build:frontend": "vite build",

"build:backend": "esbuild server/index.ts --bundle --platform=node --target=node18 --outfile=dist/server.js --external:@neondatabase/serverless --external:ws",

"preview": "vite preview",

"db:generate": "drizzle-kit generate",

"db:push": "drizzle-kit push",

"db:migrate": "drizzle-kit migrate",

"db:studio": "drizzle-kit studio"

},

"dependencies": {

"@hookform/resolvers": "^3.3.4",

"@neondatabase/serverless": "^0.9.0",

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

"@radix-ui/react-alert-dialog": "^1.0.5",

"@radix-ui/react-aspect-ratio": "^1.0.3",

"@radix-ui/react-avatar": "^1.0.4",

"@radix-ui/react-checkbox": "^1.0.4",

"@radix-ui/react-collapsible": "^1.0.3",

"@radix-ui/react-context-menu": "^2.1.5",

"@radix-ui/react-dialog": "^1.0.5",

"@radix-ui/react-dropdown-menu": "^2.0.6",

"@radix-ui/react-hover-card": "^1.0.7",

"@radix-ui/react-label": "^2.0.2",

"@radix-ui/react-menubar": "^1.0.4",

"@radix-ui/react-navigation-menu": "^1.1.4",

"@radix-ui/react-popover": "^1.0.7",

"@radix-ui/react-progress": "^1.0.3",

"@radix-ui/react-radio-group": "^1.1.3",

"@radix-ui/react-scroll-area": "^1.0.5",

"@radix-ui/react-select": "^2.0.0",

"@radix-ui/react-separator": "^1.0.3",

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

"@radix-ui/react-slot": "^1.0.2",

"@radix-ui/react-switch": "^1.0.3",

"@radix-ui/react-tabs": "^1.0.4",

"@radix-ui/react-toast": "^1.1.5",

"@radix-ui/react-toggle": "^1.0.3",

"@radix-ui/react-toggle-group": "^1.0.4",

"@radix-ui/react-tooltip": "^1.0.7",

"@tanstack/react-query": "^5.28.6",

"@types/express": "^4.17.21",

"@types/node": "^20.11.30",

"@types/react": "^18.2.66",

"@types/react-dom": "^18.2.22",

"@vitejs/plugin-react": "^4.2.1",

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

"clsx": "^2.1.0",

"cmdk": "^1.0.0",

"date-fns": "^3.6.0",

"drizzle-orm": "^0.30.1",

"drizzle-zod": "^0.5.1",

"embla-carousel-react": "^8.0.0",

"express": "^4.19.2",

"framer-motion": "^11.0.14",

"input-otp": "^1.2.4",

"lucide-react": "^0.363.0",

"next-themes": "^0.3.0",

"react": "^18.2.0",

"react-day-picker": "^8.10.0",

"react-dom": "^18.2.0",

"react-hook-form": "^7.51.1",

"react-resizable-panels": "^2.0.16",

"recharts": "^2.12.2",

"tailwind-merge": "^2.2.2",

"tailwindcss": "^3.4.1",

"tailwindcss-animate": "^1.0.7",

"tsx": "^4.7.1",

"typescript": "^5.2.2",

"vaul": "^0.9.0",

"vite": "^5.1.6",

"wouter": "^3.0.1",

"zod": "^3.22.4"

},

"devDependencies": {

"autoprefixer": "^10.4.19",

"drizzle-kit": "^0.20.14",

"esbuild": "^0.20.2",

"postcss": "^8.4.38",

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

}

}

```

## Database Schema (shared/schema.ts)

```typescript

import {

pgTable,

serial,

text,

varchar,

integer,

timestamp,

boolean,

decimal,

date,

jsonb,

} from "drizzle-orm/pg-core";

import { createInsertSchema } from "drizzle-zod";

import { z } from "zod";

// Users table (admin authentication)

export const users = pgTable("users", {

id: serial("id").primaryKey(),

username: varchar("username", { length: 100 }).notNull().unique(),

password: varchar("password", { length: 255 }).notNull(),

role: varchar("role", { length: 50 }).notNull().default("admin"),

});

// Staff table

export const staff = pgTable("staff", {

id: serial("id").primaryKey(),

name: varchar("name", { length: 100 }).notNull(),

role: varchar("role", { length: 100 }).notNull(),

status: varchar("status", { length: 50 }).notNull().default("clocked\_out"),

pin: varchar("pin", { length: 4 }).notNull().unique(),

profilePhoto: varchar("profile\_photo", { length: 500 }),

clockInTime: timestamp("clock\_in\_time"),

breakStartTime: timestamp("break\_start\_time"),

lastClockOut: timestamp("last\_clock\_out"),

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

updatedAt: timestamp("updated\_at").defaultNow(),

});

// Clients table

export const clients = pgTable("clients", {

id: serial("id").primaryKey(),

name: varchar("name", { length: 100 }).notNull(),

email: varchar("email", { length: 255 }).notNull().unique(),

phone: varchar("phone", { length: 20 }).notNull(),

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

password: varchar("password", { length: 255 }).notNull(),

emergencyContactName: varchar("emergency\_contact\_name", { length: 100 }).notNull(),

emergencyContactPhone: varchar("emergency\_contact\_phone", { length: 20 }).notNull(),

emergencyContactRelationship: varchar("emergency\_contact\_relationship", { length: 100 }).notNull(),

vetName: varchar("vet\_name", { length: 100 }).notNull(),

vetPhone: varchar("vet\_phone", { length: 20 }).notNull(),

vetAddress: text("vet\_address").notNull(),

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

updatedAt: timestamp("updated\_at").defaultNow(),

});

// Dogs table

export const dogs = pgTable("dogs", {

id: serial("id").primaryKey(),

clientId: integer("client\_id").notNull().references(() => clients.id),

name: varchar("name", { length: 100 }).notNull(),

breed: varchar("breed", { length: 100 }).notNull(),

age: integer("age"),

weight: varchar("weight", { length: 50 }),

photo: varchar("photo", { length: 500 }),

feedingInstructions: text("feeding\_instructions"),

feedingTimes: jsonb("feeding\_times"),

medication: text("medication"),

allergies: text("allergies"),

behaviorNotes: text("behavior\_notes"),

specialCare: text("special\_care"),

vetInfo: text("vet\_info"),

microchipNumber: varchar("microchip\_number", { length: 50 }),

emergencyContact: text("emergency\_contact"),

exerciseRequirements: text("exercise\_requirements"),

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

updatedAt: timestamp("updated\_at").defaultNow(),

});

// Kennels table

export const kennels = pgTable("kennels", {

id: serial("id").primaryKey(),

number: integer("number").notNull().unique(),

status: varchar("status", { length: 50 }).notNull().default("available"),

dogId: integer("dog\_id").references(() => dogs.id),

dogIds: integer("dog\_ids").array(),

checkInDate: timestamp("check\_in\_date"),

checkOutDate: timestamp("check\_out\_date"),

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

updatedAt: timestamp("updated\_at").defaultNow(),

});

// Jobs table

export const jobs = pgTable("jobs", {

id: serial("id").primaryKey(),

title: varchar("title", { length: 200 }).notNull(),

description: text("description"),

assignedTo: integer("assigned\_to").references(() => staff.id),

status: varchar("status", { length: 50 }).notNull().default("pending"),

priority: varchar("priority", { length: 20 }).notNull().default("medium"),

scheduledDate: date("scheduled\_date"),

scheduledTime: varchar("scheduled\_time", { length: 10 }),

completedAt: timestamp("completed\_at"),

notes: text("notes"),

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

updatedAt: timestamp("updated\_at").defaultNow(),

});

// Bookings table

export const bookings = pgTable("bookings", {

id: serial("id").primaryKey(),

clientId: integer("client\_id").notNull().references(() => clients.id),

dogId: integer("dog\_id").references(() => dogs.id),

serviceType: varchar("service\_type", { length: 100 }).notNull(),

checkInDate: date("check\_in\_date").notNull(),

checkOutDate: date("check\_out\_date").notNull(),

status: varchar("status", { length: 50 }).notNull().default("confirmed"),

notes: text("notes"),

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

updatedAt: timestamp("updated\_at").defaultNow(),

isActive: boolean("is\_active").notNull().default(true),

});

// Daily Reports table

export const dailyReports = pgTable("daily\_reports", {

id: serial("id").primaryKey(),

dogId: integer("dog\_id").notNull().references(() => dogs.id),

reportDate: date("report\_date").notNull(),

feeding: text("feeding"),

exercise: text("exercise"),

behavior: text("behavior"),

medication: text("medication"),

notes: text("notes"),

staffId: integer("staff\_id").references(() => staff.id),

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

updatedAt: timestamp("updated\_at").defaultNow(),

});

// Invoices table

export const invoices = pgTable("invoices", {

id: serial("id").primaryKey(),

clientId: integer("client\_id").notNull().references(() => clients.id),

amount: decimal("amount", { precision: 10, scale: 2 }).notNull(),

balanceRemaining: decimal("balance\_remaining", { precision: 10, scale: 2 }).notNull(),

status: varchar("status", { length: 50 }).notNull().default("pending"),

issueDate: timestamp("issue\_date").notNull(),

dueDate: timestamp("due\_date"),

paidDate: timestamp("paid\_date"),

description: text("description"),

estimateId: integer("estimate\_id"),

bookingId: integer("booking\_id").references(() => bookings.id),

paymentMethod: varchar("payment\_method", { length: 50 }),

transactionId: varchar("transaction\_id", { length: 100 }),

discount: decimal("discount", { precision: 10, scale: 2 }),

tax: decimal("tax", { precision: 10, scale: 2 }),

services: text("services"),

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

updatedAt: timestamp("updated\_at").defaultNow(),

});

// Time Entries table

export const timeEntries = pgTable("time\_entries", {

id: serial("id").primaryKey(),

staffId: integer("staff\_id").notNull().references(() => staff.id),

date: date("date").notNull(),

clockInTime: timestamp("clock\_in\_time").notNull(),

clockOutTime: timestamp("clock\_out\_time"),

breakStartTime: timestamp("break\_start\_time"),

breakEndTime: timestamp("break\_end\_time"),

totalBreakTime: integer("total\_break\_time"),

notes: text("notes"),

});

// Kennel Logs table

export const kennelLogs = pgTable("kennel\_logs", {

id: serial("id").primaryKey(),

kennelId: integer("kennel\_id").notNull().references(() => kennels.id),

staffId: integer("staff\_id").notNull().references(() => staff.id),

dogId: integer("dog\_id").references(() => dogs.id),

activityType: varchar("activity\_type", { length: 100 }).notNull(),

scheduledTime: timestamp("scheduled\_time"),

notes: text("notes"),

completed: boolean("completed").default(false),

});

// Service Pricing table

export const servicePricing = pgTable("service\_pricing", {

id: serial("id").primaryKey(),

serviceType: varchar("service\_type", { length: 100 }).notNull(),

serviceName: varchar("service\_name", { length: 100 }).notNull(),

pricePerUnit: decimal("price\_per\_unit", { precision: 10, scale: 2 }).notNull(),

unit: varchar("unit", { length: 50 }).notNull(),

description: text("description"),

isActive: boolean("is\_active").default(true),

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

updatedAt: timestamp("updated\_at").defaultNow(),

});

// Estimates table

export const estimates = pgTable("estimates", {

id: serial("id").primaryKey(),

clientId: integer("client\_id").notNull().references(() => clients.id),

bookingId: integer("booking\_id").references(() => bookings.id),

kennelNumber: integer("kennel\_number"),

dogNames: text("dog\_names"),

services: text("services"),

checkInDate: date("check\_in\_date"),

checkOutDate: date("check\_out\_date"),

totalDays: integer("total\_days"),

totalCost: decimal("total\_cost", { precision: 10, scale: 2 }),

status: varchar("status", { length: 20 }).default("pending"),

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

updatedAt: timestamp("updated\_at").defaultNow(),

});

// Insert schemas

export const insertUserSchema = createInsertSchema(users).omit({ id: true });

export const insertStaffSchema = createInsertSchema(staff).omit({ id: true, createdAt: true, updatedAt: true });

export const insertClientSchema = createInsertSchema(clients).omit({ id: true, createdAt: true, updatedAt: true });

export const insertDogSchema = createInsertSchema(dogs).omit({ id: true, createdAt: true, updatedAt: true });

export const insertKennelSchema = createInsertSchema(kennels).omit({ id: true, createdAt: true, updatedAt: true });

export const insertJobSchema = createInsertSchema(jobs).omit({ id: true, createdAt: true, updatedAt: true });

export const insertBookingSchema = createInsertSchema(bookings).omit({ id: true, createdAt: true, updatedAt: true });

export const insertDailyReportSchema = createInsertSchema(dailyReports).omit({ id: true, createdAt: true, updatedAt: true });

export const insertInvoiceSchema = createInsertSchema(invoices).omit({ id: true, createdAt: true, updatedAt: true });

export const insertTimeEntrySchema = createInsertSchema(timeEntries).omit({ id: true });

export const insertKennelLogSchema = createInsertSchema(kennelLogs).omit({ id: true });

export const insertServicePricingSchema = createInsertSchema(servicePricing).omit({ id: true, createdAt: true, updatedAt: true });

export const insertEstimateSchema = createInsertSchema(estimates).omit({ id: true, createdAt: true, updatedAt: true });

// Types

export type User = typeof users.$inferSelect;

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

export type Staff = typeof staff.$inferSelect;

export type InsertStaff = z.infer<typeof insertStaffSchema>;

export type Client = typeof clients.$inferSelect;

export type InsertClient = z.infer<typeof insertClientSchema>;

export type Dog = typeof dogs.$inferSelect;

export type InsertDog = z.infer<typeof insertDogSchema>;

export type Kennel = typeof kennels.$inferSelect;

export type InsertKennel = z.infer<typeof insertKennelSchema>;

export type Job = typeof jobs.$inferSelect;

export type InsertJob = z.infer<typeof insertJobSchema>;

export type Booking = typeof bookings.$inferSelect;

export type InsertBooking = z.infer<typeof insertBookingSchema>;

export type DailyReport = typeof dailyReports.$inferSelect;

export type InsertDailyReport = z.infer<typeof insertDailyReportSchema>;

export type Invoice = typeof invoices.$inferSelect;

export type InsertInvoice = z.infer<typeof insertInvoiceSchema>;

export type TimeEntry = typeof timeEntries.$inferSelect;

export type InsertTimeEntry = z.infer<typeof insertTimeEntrySchema>;

export type KennelLog = typeof kennelLogs.$inferSelect;

export type InsertKennelLog = z.infer<typeof insertKennelLogSchema>;

export type ServicePricing = typeof servicePricing.$inferSelect;

export type InsertServicePricing = z.infer<typeof insertServicePricingSchema>;

export type Estimate = typeof estimates.$inferSelect;

export type InsertEstimate = z.infer<typeof insertEstimateSchema>;

```

## Database Connection (server/db.ts)

```typescript

import { Pool, neonConfig } from '@neondatabase/serverless';

import { drizzle } from 'drizzle-orm/neon-serverless';

import ws from "ws";

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

neonConfig.webSocketConstructor = ws;

if (!process.env.DATABASE\_URL) {

throw new Error(

"DATABASE\_URL must be set. Did you forget to provision a database?",

);

}

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

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

```

## Storage Interface (server/storage-clean.ts)

```typescript

import type {

User,

InsertUser,

Staff,

InsertStaff,

Client,

InsertClient,

Dog,

InsertDog,

Kennel,

InsertKennel,

Job,

InsertJob,

Booking,

InsertBooking,

DailyReport,

InsertDailyReport,

Invoice,

InsertInvoice,

TimeEntry,

InsertTimeEntry,

KennelLog,

InsertKennelLog,

ServicePricing,

InsertServicePricing,

Estimate,

InsertEstimate,

} from "@shared/schema";

// Interface for storage operations

export interface IStorage {

// Users

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

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

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

// Staff

getAllStaff(): Promise<Staff[]>;

getStaff(id: number): Promise<Staff | undefined>;

getStaffByPin(pin: string): Promise<Staff | undefined>;

createStaff(staff: InsertStaff): Promise<Staff>;

updateStaff(id: number, updates: Partial<Staff>): Promise<Staff | undefined>;

// Clients

getAllClients(): Promise<Client[]>;

getClient(id: number): Promise<Client | undefined>;

getClientByEmail(email: string): Promise<Client | undefined>;

createClient(client: InsertClient): Promise<Client>;

updateClient(id: number, updates: Partial<Client>): Promise<Client | undefined>;

deleteClient(id: number): Promise<boolean>;

// Dogs

getAllDogs(): Promise<Dog[]>;

getDog(id: number): Promise<Dog | undefined>;

getDogsByClient(clientId: number): Promise<Dog[]>;

createDog(dog: InsertDog): Promise<Dog>;

updateDog(id: number, updates: Partial<Dog>): Promise<Dog | undefined>;

deleteDog(id: number): Promise<boolean>;

// Kennels

getAllKennels(): Promise<Kennel[]>;

getKennel(id: number): Promise<Kennel | undefined>;

getKennelByNumber(number: number): Promise<Kennel | undefined>;

createKennel(kennel: InsertKennel): Promise<Kennel>;

updateKennel(id: number, updates: Partial<Kennel>): Promise<Kennel | undefined>;

// Jobs

getAllJobs(): Promise<Job[]>;

getJob(id: number): Promise<Job | undefined>;

getJobsByStaff(staffId: number): Promise<Job[]>;

getJobsByDate(date: string): Promise<Job[]>;

createJob(job: InsertJob): Promise<Job>;

updateJob(id: number, updates: Partial<Job>): Promise<Job | undefined>;

// Bookings

getAllBookings(): Promise<Booking[]>;

getBooking(id: number): Promise<Booking | undefined>;

getBookingsByClient(clientId: number): Promise<Booking[]>;

createBooking(booking: InsertBooking): Promise<Booking>;

updateBooking(id: number, updates: Partial<Booking>): Promise<Booking | undefined>;

deleteBooking(id: number): Promise<boolean>;

// Daily Reports

getDailyReport(dogId: number, date: string): Promise<DailyReport | undefined>;

createDailyReport(report: InsertDailyReport): Promise<DailyReport>;

updateDailyReport(id: number, updates: Partial<DailyReport>): Promise<DailyReport | undefined>;

// Invoices

getAllInvoices(): Promise<Invoice[]>;

getInvoice(id: number): Promise<Invoice | undefined>;

getInvoicesByClient(clientId: number): Promise<Invoice[]>;

createInvoice(invoice: InsertInvoice): Promise<Invoice>;

updateInvoice(id: number, updates: Partial<Invoice>): Promise<Invoice | undefined>;

deleteInvoice(id: number): Promise<boolean>;

// Estimates

getAllEstimates(): Promise<any[]>;

getEstimate(id: number): Promise<any | undefined>;

createEstimate(estimate: any): Promise<any>;

deleteEstimate(id: number): Promise<boolean>;

// Time Entries

getAllTimeEntries(): Promise<TimeEntry[]>;

getTimeEntry(id: number): Promise<TimeEntry | undefined>;

getTimeEntriesByStaff(staffId: number): Promise<TimeEntry[]>;

getTimeEntriesByDate(date: string): Promise<TimeEntry[]>;

createTimeEntry(timeEntry: InsertTimeEntry): Promise<TimeEntry>;

updateTimeEntry(id: number, updates: Partial<TimeEntry>): Promise<TimeEntry | undefined>;

// Kennel Logs

getAllKennelLogs(): Promise<KennelLog[]>;

getKennelLog(id: number): Promise<KennelLog | undefined>;

getKennelLogsByKennel(kennelId: number): Promise<KennelLog[]>;

getKennelLogsByDog(dogId: number): Promise<KennelLog[]>;

getKennelLogsByStaff(staffId: number): Promise<KennelLog[]>;

getKennelLogsByDate(date: string): Promise<KennelLog[]>;

createKennelLog(kennelLog: InsertKennelLog): Promise<KennelLog>;

updateKennelLog(id: number, updates: Partial<KennelLog>): Promise<KennelLog | undefined>;

// Service Pricing

getAllServicePricing(): Promise<ServicePricing[]>;

getServicePricing(id: number): Promise<ServicePricing | undefined>;

getServicePricingByType(serviceType: string): Promise<ServicePricing | undefined>;

createServicePricing(pricing: InsertServicePricing): Promise<ServicePricing>;

updateServicePricing(id: number, updates: Partial<ServicePricing>): Promise<ServicePricing | undefined>;

deleteServicePricing(id: number): Promise<boolean>;

}

// In-memory storage implementation for demo

class CleanMemStorage implements IStorage {

private users: Map<number, User> = new Map();

private staff: Map<number, Staff> = new Map();

private clients: Map<number, Client> = new Map();

private dogs: Map<number, Dog> = new Map();

private kennels: Map<number, Kennel> = new Map();

private jobs: Map<number, Job> = new Map();

private bookings: Map<number, Booking> = new Map();

private dailyReports: Map<number, DailyReport> = new Map();

private invoices: Map<number, Invoice> = new Map();

private timeEntries: Map<number, TimeEntry> = new Map();

private kennelLogs: Map<number, KennelLog> = new Map();

private servicePricing: Map<number, ServicePricing> = new Map();

private estimates: Map<number, any> = new Map();

private currentId: number = 1;

constructor() {

this.initializeAdmin();

this.initializeServicePricing();

}

private initializeAdmin() {

const adminUser: User = {

id: this.currentId++,

username: "admin",

password: "admin123",

role: "admin",

};

this.users.set(adminUser.id, adminUser);

// Initialize staff

const staff1: Staff = {

id: this.currentId++,

name: "Emma Thompson",

role: "Head Trainer",

status: "clocked\_out",

pin: "1234",

profilePhoto: null,

clockInTime: null,

breakStartTime: null,

lastClockOut: null,

createdAt: new Date(),

updatedAt: new Date(),

};

const staff2: Staff = {

id: this.currentId++,

name: "James Wilson",

role: "Dog Walker",

status: "clocked\_out",

pin: "5678",

profilePhoto: null,

clockInTime: null,

breakStartTime: null,

lastClockOut: null,

createdAt: new Date(),

updatedAt: new Date(),

};

const staff3: Staff = {

id: this.currentId++,

name: "Sarah Johnson",

role: "Kennel Assistant",

status: "clocked\_out",

pin: "9012",

profilePhoto: null,

clockInTime: null,

breakStartTime: null,

lastClockOut: null,

createdAt: new Date(),

updatedAt: new Date(),

};

this.staff.set(staff1.id, staff1);

this.staff.set(staff2.id, staff2);

this.staff.set(staff3.id, staff3);

// Initialize kennels

for (let i = 1; i <= 20; i++) {

const kennel: Kennel = {

id: this.currentId++,

number: i,

status: "available",

dogId: null,

dogIds: null,

checkInDate: null,

checkOutDate: null,

createdAt: new Date(),

updatedAt: new Date(),

};

this.kennels.set(kennel.id, kennel);

}

}

private initializeServicePricing() {

const services = [

{ serviceType: "boarding", serviceName: "Dog Boarding", pricePerUnit: 40.00, unit: "night" },

{ serviceType: "training", serviceName: "Group Training", pricePerUnit: 40.00, unit: "session" },

{ serviceType: "training", serviceName: "1-on-1 Training", pricePerUnit: 45.00, unit: "session" },

{ serviceType: "walking", serviceName: "Dog Walking", pricePerUnit: 20.00, unit: "walk" },

];

services.forEach(service => {

const servicePricing: ServicePricing = {

id: this.currentId++,

serviceType: service.serviceType,

serviceName: service.serviceName,

pricePerUnit: service.pricePerUnit.toString(),

unit: service.unit,

description: null,

isActive: true,

createdAt: new Date(),

updatedAt: new Date(),

};

this.servicePricing.set(servicePricing.id, servicePricing);

});

}

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

return this.users.get(id);

}

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

return Array.from(this.users.values()).find(user => user.username === username);

}

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

const id = this.currentId++;

const user: User = { ...insertUser, id };

this.users.set(id, user);

return user;

}

async getAllStaff(): Promise<Staff[]> {

return Array.from(this.staff.values());

}

async getStaff(id: number): Promise<Staff | undefined> {

return this.staff.get(id);

}

async getStaffByPin(pin: string): Promise<Staff | undefined> {

return Array.from(this.staff.values()).find(staff => staff.pin === pin);

}

async createStaff(insertStaff: InsertStaff): Promise<Staff> {

const id = this.currentId++;

const staff: Staff = {

...insertStaff,

id,

createdAt: new Date(),

updatedAt: new Date(),

};

this.staff.set(id, staff);

return staff;

}

async updateStaff(id: number, updates: Partial<Staff>): Promise<Staff | undefined> {

const staff = this.staff.get(id);

if (!staff) return undefined;

const updatedStaff = { ...staff, ...updates, updatedAt: new Date() };

this.staff.set(id, updatedStaff);

return updatedStaff;

}

async getAllClients(): Promise<Client[]> {

return Array.from(this.clients.values());

}

async getClient(id: number): Promise<Client | undefined> {

return this.clients.get(id);

}

async getClientByEmail(email: string): Promise<Client | undefined> {

return Array.from(this.clients.values()).find(client => client.email === email);

}

async createClient(insertClient: InsertClient): Promise<Client> {

const id = this.currentId++;

const client: Client = {

...insertClient,

id,

createdAt: new Date(),

updatedAt: new Date(),

};

this.clients.set(id, client);

return client;

}

async updateClient(id: number, updates: Partial<Client>): Promise<Client | undefined> {

const client = this.clients.get(id);

if (!client) return undefined;

const updatedClient = { ...client, ...updates, updatedAt: new Date() };

this.clients.set(id, updatedClient);

return updatedClient;

}

async deleteClient(id: number): Promise<boolean> {

return this.clients.delete(id);

}

async getAllDogs(): Promise<Dog[]> {

return Array.from(this.dogs.values());

}

async getDog(id: number): Promise<Dog | undefined> {

return this.dogs.get(id);

}

async getDogsByClient(clientId: number): Promise<Dog[]> {

return Array.from(this.dogs.values()).filter(dog => dog.clientId === clientId);

}

async createDog(insertDog: InsertDog): Promise<Dog> {

const id = this.currentId++;

const dog: Dog = {

...insertDog,

id,

createdAt: new Date(),

updatedAt: new Date(),

};

this.dogs.set(id, dog);

return dog;

}

async updateDog(id: number, updates: Partial<Dog>): Promise<Dog | undefined> {

const dog = this.dogs.get(id);

if (!dog) return undefined;

const updatedDog = { ...dog, ...updates, updatedAt: new Date() };

this.dogs.set(id, updatedDog);

return updatedDog;

}

async deleteDog(id: number): Promise<boolean> {

return this.dogs.delete(id);

}

async getAllKennels(): Promise<Kennel[]> {

return Array.from(this.kennels.values());

}

async getKennel(id: number): Promise<Kennel | undefined> {

return this.kennels.get(id);

}

async getKennelByNumber(number: number): Promise<Kennel | undefined> {

return Array.from(this.kennels.values()).find(kennel => kennel.number === number);

}

async createKennel(insertKennel: InsertKennel): Promise<Kennel> {

const id = this.currentId++;

const kennel: Kennel = {

...insertKennel,

id,

createdAt: new Date(),

updatedAt: new Date(),

};

this.kennels.set(id, kennel);

return kennel;

}

async updateKennel(id: number, updates: Partial<Kennel>): Promise<Kennel | undefined> {

const kennel = this.kennels.get(id);

if (!kennel) return undefined;

const updatedKennel = { ...kennel, ...updates, updatedAt: new Date() };

this.kennels.set(id, updatedKennel);

return updatedKennel;

}

async getAllJobs(): Promise<Job[]> {

return Array.from(this.jobs.values());

}

async getJob(id: number): Promise<Job | undefined> {

return this.jobs.get(id);

}

async getJobsByStaff(staffId: number): Promise<Job[]> {

return Array.from(this.jobs.values()).filter(job => job.assignedTo === staffId);

}

async getJobsByDate(date: string): Promise<Job[]> {

return Array.from(this.jobs.values()).filter(job => job.scheduledDate === date);

}

async createJob(insertJob: InsertJob): Promise<Job> {

const id = this.currentId++;

const job: Job = {

...insertJob,

id,

createdAt: new Date(),

updatedAt: new Date(),

};

this.jobs.set(id, job);

return job;

}

async updateJob(id: number, updates: Partial<Job>): Promise<Job | undefined> {

const job = this.jobs.get(id);

if (!job) return undefined;

const updatedJob = { ...job, ...updates, updatedAt: new Date() };

this.jobs.set(id, updatedJob);

return updatedJob;

}

async getAllBookings(): Promise<Booking[]> {

return Array.from(this.bookings.values());

}

async getBooking(id: number): Promise<Booking | undefined> {

return this.bookings.get(id);

}

async getBookingsByClient(clientId: number): Promise<Booking[]> {

return Array.from(this.bookings.values()).filter(booking => booking.clientId === clientId);

}

async createBooking(insertBooking: InsertBooking): Promise<Booking> {

const id = this.currentId++;

const booking: Booking = {

...insertBooking,

id,

createdAt: new Date(),

updatedAt: new Date(),

};

this.bookings.set(id, booking);

return booking;

}

async updateBooking(id: number, updates: Partial<Booking>): Promise<Booking | undefined> {

const booking = this.bookings.get(id);

if (!booking) return undefined;

const updatedBooking = { ...booking, ...updates, updatedAt: new Date() };

this.bookings.set(id, updatedBooking);

return updatedBooking;

}

async deleteBooking(id: number): Promise<boolean> {

return this.bookings.delete(id);

}

async getDailyReport(dogId: number, date: string): Promise<DailyReport | undefined> {

return Array.from(this.dailyReports.values()).find(

report => report.dogId === dogId && report.reportDate === date

);

}

async createDailyReport(insertReport: InsertDailyReport): Promise<DailyReport> {

const id = this.currentId++;

const report: DailyReport = {

...insertReport,

id,

createdAt: new Date(),

updatedAt: new Date(),

};

this.dailyReports.set(id, report);

return report;

}

async updateDailyReport(id: number, updates: Partial<DailyReport>): Promise<DailyReport | undefined> {

const report = this.dailyReports.get(id);

if (!report) return undefined;

const updatedReport = { ...report, ...updates, updatedAt: new Date() };

this.dailyReports.set(id, updatedReport);

return updatedReport;

}

async getAllInvoices(): Promise<Invoice[]> {

return Array.from(this.invoices.values());

}

async getInvoice(id: number): Promise<Invoice | undefined> {

return this.invoices.get(id);

}

async getInvoicesByClient(clientId: number): Promise<Invoice[]> {

return Array.from(this.invoices.values()).filter(invoice => invoice.clientId === clientId);

}

async createInvoice(insertInvoice: InsertInvoice): Promise<Invoice> {

const id = this.currentId++;

const invoice: Invoice = {

...insertInvoice,

id,

createdAt: new Date(),

updatedAt: new Date(),

};

this.invoices.set(id, invoice);

return invoice;

}

async updateInvoice(id: number, updates: Partial<Invoice>): Promise<Invoice | undefined> {

const invoice = this.invoices.get(id);

if (!invoice) return undefined;

const updatedInvoice = { ...invoice, ...updates, updatedAt: new Date() };

this.invoices.set(id, updatedInvoice);

return updatedInvoice;

}

async deleteInvoice(id: number): Promise<boolean> {

return this.invoices.delete(id);

}

async getAllEstimates(): Promise<any[]> {

return Array.from(this.estimates.values());

}

async getEstimate(id: number): Promise<any | undefined> {

return this.estimates.get(id);

}

async createEstimate(estimate: any): Promise<any> {

const id = this.currentId++;

const newEstimate = { ...estimate, id };

this.estimates.set(id, newEstimate);

return newEstimate;

}

async deleteEstimate(id: number): Promise<boolean> {

console.log(`🗑️ CleanMemStorage: Attempting to delete estimate with ID: ${id}`);

const existed = this.estimates.has(id);

const deleted = this.estimates.delete(id);

console.log(`🗑️ CleanMemStorage: Estimate ${id} existed: ${existed}, deleted: ${deleted}`);

return deleted;

}

async getAllTimeEntries(): Promise<TimeEntry[]> {

return Array.from(this.timeEntries.values());

}

async getTimeEntry(id: number): Promise<TimeEntry | undefined> {

return this.timeEntries.get(id);

}

async getTimeEntriesByStaff(staffId: number): Promise<TimeEntry[]> {

return Array.from(this.timeEntries.values()).filter(entry => entry.staffId === staffId);

}

async getTimeEntriesByDate(date: string): Promise<TimeEntry[]> {

return Array.from(this.timeEntries.values()).filter(entry => entry.date === date);

}

async createTimeEntry(insertTimeEntry: InsertTimeEntry): Promise<TimeEntry> {

const id = this.currentId++;

const timeEntry: TimeEntry = {

...insertTimeEntry,

id,

};

this.timeEntries.set(id, timeEntry);

return timeEntry;

}

async updateTimeEntry(id: number, updates: Partial<TimeEntry>): Promise<TimeEntry | undefined> {

const timeEntry = this.timeEntries.get(id);

if (!timeEntry) return undefined;

const updatedTimeEntry = { ...timeEntry, ...updates };

this.timeEntries.set(id, updatedTimeEntry);

return updatedTimeEntry;

}

async getAllKennelLogs(): Promise<KennelLog[]> {

return Array.from(this.kennelLogs.values());

}

async getKennelLog(id: number): Promise<KennelLog | undefined> {

return this.kennelLogs.get(id);

}

async getKennelLogsByKennel(kennelId: number): Promise<KennelLog[]> {

return Array.from(this.kennelLogs.values()).filter(log => log.kennelId === kennelId);

}

async getKennelLogsByDog(dogId: number): Promise<KennelLog[]> {

return Array.from(this.kennelLogs.values()).filter(log => log.dogId === dogId);

}

async getKennelLogsByStaff(staffId: number): Promise<KennelLog[]> {

return Array.from(this.kennelLogs.values()).filter(log => log.staffId === staffId);

}

async getKennelLogsByDate(date: string): Promise<KennelLog[]> {

return Array.from(this.kennelLogs.values()).filter(log =>

log.scheduledTime && log.scheduledTime.toISOString().split('T')[0] === date

);

}

async createKennelLog(insertKennelLog: InsertKennelLog): Promise<KennelLog> {

const id = this.currentId++;

const kennelLog: KennelLog = {

...insertKennelLog,

id,

};

this.kennelLogs.set(id, kennelLog);

return kennelLog;

}

async updateKennelLog(id: number, updates: Partial<KennelLog>): Promise<KennelLog | undefined> {

const kennelLog = this.kennelLogs.get(id);

if (!kennelLog) return undefined;

const updatedKennelLog = { ...kennelLog, ...updates };

this.kennelLogs.set(id, updatedKennelLog);

return updatedKennelLog;

}

async getAllServicePricing(): Promise<ServicePricing[]> {

return Array.from(this.servicePricing.values());

}

async getServicePricing(id: number): Promise<ServicePricing | undefined> {

return this.servicePricing.get(id);

}

async getServicePricingByType(serviceType: string): Promise<ServicePricing | undefined> {

return Array.from(this.servicePricing.values()).find(pricing => pricing.serviceType === serviceType);

}

async createServicePricing(insertPricing: InsertServicePricing): Promise<ServicePricing> {

const id = this.currentId++;

const pricing: ServicePricing = {

...insertPricing,

id,

createdAt: new Date(),

updatedAt: new Date(),

};

this.servicePricing.set(id, pricing);

return pricing;

}

async updateServicePricing(id: number, updates: Partial<ServicePricing>): Promise<ServicePricing | undefined> {

const pricing = this.servicePricing.get(id);

if (!pricing) return undefined;

const updatedPricing = { ...pricing, ...updates, updatedAt: new Date() };

this.servicePricing.set(id, updatedPricing);

return updatedPricing;

}

async deleteServicePricing(id: number): Promise<boolean> {

return this.servicePricing.delete(id);

}

}

export const storage = new CleanMemStorage();

```

## Main Server Routes (server/routes.ts)

```typescript

import type { Express } from "express";

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

import { storage } from "./storage";

import { securityMiddleware } from "./security";

import { insertStaffSchema, insertClientSchema, insertDogSchema, insertJobSchema, insertBookingSchema } from "@shared/schema";

import { z } from "zod";

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

// Authentication routes

app.post("/api/auth/login", async (req, res) => {

try {

const { username, password } = req.body;

const user = await storage.getUserByUsername(username);

if (!user || user.password !== password) {

return res.status(401).json({ message: "Invalid credentials" });

}

res.json({

user: {

id: user.id,

username: user.username,

role: user.role

}

});

} catch (error) {

res.status(500).json({ message: "Internal server error" });

}

});

// Staff authentication

app.post("/api/auth/staff", async (req, res) => {

try {

const { pin } = req.body;

const staff = await storage.getStaffByPin(pin);

if (!staff) {

return res.status(401).json({ message: "Invalid PIN" });

}

res.json({ staff });

} catch (error) {

res.status(500).json({ message: "Internal server error" });

}

});

// Staff routes

app.get("/api/staff", async (req, res) => {

try {

const staff = await storage.getAllStaff();

res.json(staff);

} catch (error) {

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

}

});

app.post("/api/staff", async (req, res) => {

try {

const validation = insertStaffSchema.safeParse(req.body);

if (!validation.success) {

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

}

const staff = await storage.createStaff(validation.data);

res.status(201).json(staff);

} catch (error) {

res.status(500).json({ message: "Failed to create staff member" });

}

});

app.patch("/api/staff/:id", async (req, res) => {

try {

const id = parseInt(req.params.id);

const updates = req.body;

const staff = await storage.updateStaff(id, updates);

if (!staff) {

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

}

res.json(staff);

} catch (error) {

res.status(500).json({ message: "Failed to update staff member" });

}

});

// Client routes

app.get("/api/clients", async (req, res) => {

try {

console.log("🚀 API: Fetching clients...");

const clients = await storage.getAllClients();

console.log(`🎯 API: Got clients: ${clients.length}`, clients.map(c => c.name));

res.json(clients);

} catch (error) {

console.error("❌ API: Error fetching clients:", error);

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

}

});

app.post("/api/clients", async (req, res) => {

try {

const validation = insertClientSchema.safeParse(req.body);

if (!validation.success) {

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

}

const client = await storage.createClient(validation.data);

res.status(201).json(client);

} catch (error) {

res.status(500).json({ message: "Failed to create client" });

}

});

app.patch("/api/clients/:id", async (req, res) => {

try {

const id = parseInt(req.params.id);

const updates = req.body;

const client = await storage.updateClient(id, updates);

if (!client) {

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

}

res.json(client);

} catch (error) {

res.status(500).json({ message: "Failed to update client" });

}

});

app.delete("/api/clients/:id", async (req, res) => {

try {

const id = parseInt(req.params.id);

const success = await storage.deleteClient(id);

if (!success) {

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

}

res.json({ message: "Client deleted successfully" });

} catch (error) {

res.status(500).json({ message: "Failed to delete client" });

}

});

// Dog routes

app.get("/api/dogs", async (req, res) => {

try {

const dogs = await storage.getAllDogs();

res.json(dogs);

} catch (error) {

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

}

});

app.post("/api/dogs", async (req, res) => {

try {

const validation = insertDogSchema.safeParse(req.body);

if (!validation.success) {

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

}

const dog = await storage.createDog(validation.data);

res.status(201).json(dog);

} catch (error) {

res.status(500).json({ message: "Failed to create dog" });

}

});

app.patch("/api/dogs/:id", async (req, res) => {

try {

const id = parseInt(req.params.id);

const updates = req.body;

const dog = await storage.updateDog(id, updates);

if (!dog) {

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

}

res.json(dog);

} catch (error) {

res.status(500).json({ message: "Failed to update dog" });

}

});

app.delete("/api/dogs/:id", async (req, res) => {

try {

const id = parseInt(req.params.id);

const success = await storage.deleteDog(id);

if (!success) {

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

}

res.json({ message: "Dog deleted successfully" });

} catch (error) {

res.status(500).json({ message: "Failed to delete dog" });

}

});

// Kennel routes

app.get("/api/kennels", async (req, res) => {

try {

const kennels = await storage.getAllKennels();

res.json(kennels);

} catch (error) {

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

}

});

app.patch("/api/kennels/:id", async (req, res) => {

try {

const id = parseInt(req.params.id);

const updates = req.body;

const kennel = await storage.updateKennel(id, updates);

if (!kennel) {

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

}

res.json(kennel);

} catch (error) {

res.status(500).json({ message: "Failed to update kennel" });

}

});

// Job routes

app.get("/api/jobs", async (req, res) => {

try {

const jobs = await storage.getAllJobs();

res.json(jobs);

} catch (error) {

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

}

});

app.post("/api/jobs", async (req, res) => {

try {

const validation = insertJobSchema.safeParse(req.body);

if (!validation.success) {

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

}

const job = await storage.createJob(validation.data);

res.status(201).json(job);

} catch (error) {

res.status(500).json({ message: "Failed to create job" });

}

});

app.patch("/api/jobs/:id", async (req, res) => {

try {

const id = parseInt(req.params.id);

const updates = req.body;

const job = await storage.updateJob(id, updates);

if (!job) {

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

}

res.json(job);

} catch (error) {

res.status(500).json({ message: "Failed to update job" });

}

});

// Booking routes

app.get("/api/bookings", async (req, res) => {

try {

const allBookings = await storage.getAllBookings();

const activeBookings = allBookings.filter(booking => booking.isActive !== false);

console.log(`📋 Returning ${activeBookings.length} active bookings out of ${allBookings.length} total`);

res.json(activeBookings);

} catch (error) {

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

}

});

app.post("/api/bookings", async (req, res) => {

try {

const validation = insertBookingSchema.safeParse(req.body);

if (!validation.success) {

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

}

const booking = await storage.createBooking(validation.data);

res.status(201).json(booking);

} catch (error) {

res.status(500).json({ message: "Failed to create booking" });

}

});

app.patch("/api/bookings/:id", async (req, res) => {

try {

const id = parseInt(req.params.id);

const updates = req.body;

const booking = await storage.updateBooking(id, updates);

if (!booking) {

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

}

res.json(booking);

} catch (error) {

res.status(500).json({ message: "Failed to update booking" });

}

});

app.delete("/api/bookings/:id", async (req, res) => {

try {

const id = parseInt(req.params.id);

const success = await storage.deleteBooking(id);

if (!success) {

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

}

res.json({ message: "Booking deleted successfully" });

} catch (error) {

res.status(500).json({ message: "Failed to delete booking" });

}

});

// Invoice routes

app.get("/api/invoices", async (req, res) => {

try {

const invoices = await storage.getAllInvoices();

res.json(invoices);

} catch (error) {

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

}

});

app.post("/api/invoices", async (req, res) => {

try {

const { clientId, amount, description } = req.body;

const invoice = await storage.createInvoice({

clientId,

amount,

status: "pending",

issueDate: new Date(),

dueDate: new Date(Date.now() + 30 \* 24 \* 60 \* 60 \* 1000),

description,

bookingId: null,

});

res.status(201).json(invoice);

} catch (error) {

res.status(500).json({ message: "Failed to create invoice" });

}

});

// Service Pricing routes

app.get("/api/service-pricing", async (req, res) => {

try {

const pricing = await storage.getAllServicePricing();

res.json(pricing);

} catch (error) {

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

}

});

// Estimates routes

app.get("/api/estimates", async (req, res) => {

try {

const estimates = await storage.getAllEstimates();

console.log(`📊 ESTIMATES: Returning ${estimates.length} estimates`);

res.json(estimates);

} catch (error) {

console.error("❌ ESTIMATES: Error fetching estimates:", error);

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

}

});

app.post("/api/estimates", async (req, res) => {

try {

const { clientId, dogIds, services, checkInDate, checkOutDate, notes } = req.body;

// Calculate total cost based on booking duration and service pricing

const servicePricing = await storage.getAllServicePricing();

const boardingPrice = servicePricing.find(p => p.serviceType === "boarding")?.pricePerUnit || "40";

const startDate = new Date(checkInDate);

const endDate = new Date(checkOutDate);

const totalDays = Math.ceil((endDate.getTime() - startDate.getTime()) / (1000 \* 60 \* 60 \* 24));

const totalCost = totalDays \* parseFloat(boardingPrice) \* dogIds.length;

// Create the booking first

const booking = await storage.createBooking({

clientId: parseInt(clientId),

dogId: dogIds[0],

serviceType: services.join(", "),

checkInDate,

checkOutDate,

status: "confirmed",

notes: notes || null,

isActive: true,

});

// Create the estimate

const estimate = await storage.createEstimate({

clientId: parseInt(clientId),

bookingId: booking.id,

kennelNumber: null,

dogNames: dogIds.map((id: number) => `Dog ${id}`).join(", "),

services: services.join(", "),

checkInDate,

checkOutDate,

totalDays,

totalCost: totalCost.toFixed(2),

status: "pending",

});

res.status(201).json({ estimate, booking });

} catch (error) {

console.error("❌ Error creating estimate:", error);

res.status(500).json({ message: "Failed to create estimate" });

}

});

// Critical fix: Ensure delete routes are properly registered

console.log("🔧 Registering delete routes...");

app.delete("/api/estimates/:id", async (req, res) => {

console.log("🎯 DELETE /api/estimates/:id route called");

try {

const estimateId = parseInt(req.params.id);

console.log(`🗑️ Route handler: Deleting estimate ${estimateId}...`);

console.log("🔄 About to call storage.deleteEstimate()");

const success = await storage.deleteEstimate(estimateId);

console.log("🔄 storage.deleteEstimate() returned:", success);

if (success) {

console.log(`✅ Successfully deleted estimate ${estimateId}`);

res.json({ message: "Estimate deleted successfully" });

} else {

console.log(`❌ Failed to delete estimate ${estimateId}`);

res.status(500).json({ message: "Failed to delete estimate" });

}

} catch (error) {

console.error("❌ Error deleting estimate:", error);

res.status(500).json({ message: "Internal server error" });

}

});

const httpServer = createServer(app);

return httpServer;

}

```

## Main Admin Dashboard (client/src/pages/simple-admin.tsx)

This is the main admin interface with estimates and booking management:

```typescript

import React, { useState, useEffect } from "react";

import { useQuery, useMutation, useQueryClient } from "@tanstack/react-query";

import { apiRequest } from "@/lib/queryClient";

import { Button } from "@/components/ui/button";

import { Card, CardContent, CardHeader, CardTitle } from "@/components/ui/card";

import { Badge } from "@/components/ui/badge";

import { Calendar, Users, Dog, DollarSign, Plus, Trash2 } from "lucide-react";

import { cn } from "@/lib/utils";

export default function SimpleAdminDashboard() {

const queryClient = useQueryClient();

// Data queries

const { data: staff } = useQuery({ queryKey: ["/api/staff"] });

const { data: clients } = useQuery({ queryKey: ["/api/clients"] });

const { data: dogs } = useQuery({ queryKey: ["/api/dogs"] });

const { data: kennels } = useQuery({ queryKey: ["/api/kennels"] });

const { data: bookings } = useQuery({ queryKey: ["/api/bookings"] });

const { data: servicePricing } = useQuery({ queryKey: ["/api/service-pricing"] });

const { data: estimates } = useQuery({ queryKey: ["/api/estimates"] });

const { data: invoices } = useQuery({ queryKey: ["/api/invoices"] });

// Delete estimate mutation

const deleteEstimateMutation = useMutation({

mutationFn: async (estimateId: number) => {

console.log(`🗑️ Frontend: Attempting to delete estimate ${estimateId}`);

const response = await apiRequest("DELETE", `/api/estimates/${estimateId}`);

if (!response.ok) {

throw new Error(`Failed to delete estimate: ${response.status}`);

}

return response.json();

},

onSuccess: () => {

console.log("✅ Frontend: Estimate deleted successfully, refreshing data...");

queryClient.invalidateQueries({ queryKey: ["/api/estimates"] });

queryClient.invalidateQueries({ queryKey: ["/api/bookings"] });

},

onError: (error) => {

console.error("❌ Frontend: Error deleting estimate:", error);

},

});

// Delete booking mutation

const deleteBookingMutation = useMutation({

mutationFn: async (bookingId: number) => {

console.log(`🗑️ Frontend: Attempting to delete booking ${bookingId}`);

const response = await apiRequest("DELETE", `/api/bookings/${bookingId}`);

if (!response.ok) {

throw new Error(`Failed to delete booking: ${response.status}`);

}

return response.json();

},

onSuccess: () => {

console.log("✅ Frontend: Booking deleted successfully, refreshing data...");

queryClient.invalidateQueries({ queryKey: ["/api/bookings"] });

queryClient.invalidateQueries({ queryKey: ["/api/estimates"] });

},

onError: (error) => {

console.error("❌ Frontend: Error deleting booking:", error);

},

});

const handleCancelEstimate = async (estimateId: number, bookingId?: number) => {

console.log(`🚫 Cancel estimate ${estimateId} with booking ${bookingId}`);

try {

// Delete estimate first

await deleteEstimateMutation.mutateAsync(estimateId);

// Delete associated booking if it exists

if (bookingId) {

await deleteBookingMutation.mutateAsync(bookingId);

}

console.log("🎉 Cancellation complete!");

} catch (error) {

console.error("❌ Cancellation failed:", error);

}

};

return (

<div className="min-h-screen bg-black text-white p-6">

<div className="max-w-7xl mx-auto">

{/\* Header \*/}

<div className="flex items-center justify-between mb-8">

<div>

<h1 className="text-4xl font-bold bg-gradient-to-r from-amber-400 to-yellow-600 bg-clip-text text-transparent">

VIP Elite K9s

</h1>

<p className="text-gray-400 mt-2">Admin Dashboard</p>

</div>

</div>

{/\* Stats Grid \*/}

<div className="grid grid-cols-1 md:grid-cols-4 gap-6 mb-8">

<Card className="bg-gray-900 border-amber-500/20">

<CardHeader className="flex flex-row items-center justify-between space-y-0 pb-2">

<CardTitle className="text-sm font-medium text-white">Total Staff</CardTitle>

<Users className="h-4 w-4 text-amber-500" />

</CardHeader>

<CardContent>

<div className="text-2xl font-bold text-amber-400">{staff?.length || 0}</div>

</CardContent>

</Card>

<Card className="bg-gray-900 border-amber-500/20">

<CardHeader className="flex flex-row items-center justify-between space-y-0 pb-2">

<CardTitle className="text-sm font-medium text-white">Active Dogs</CardTitle>

<Dog className="h-4 w-4 text-amber-500" />

</CardHeader>

<CardContent>

<div className="text-2xl font-bold text-amber-400">{dogs?.length || 0}</div>

</CardContent>

</Card>

<Card className="bg-gray-900 border-amber-500/20">

<CardHeader className="flex flex-row items-center justify-between space-y-0 pb-2">

<CardTitle className="text-sm font-medium text-white">Active Bookings</CardTitle>

<Calendar className="h-4 w-4 text-amber-500" />

</CardHeader>

<CardContent>

<div className="text-2xl font-bold text-amber-400">{bookings?.length || 0}</div>

</CardContent>

</Card>

<Card className="bg-gray-900 border-amber-500/20">

<CardHeader className="flex flex-row items-center justify-between space-y-0 pb-2">

<CardTitle className="text-sm font-medium text-white">Estimates</CardTitle>

<DollarSign className="h-4 w-4 text-amber-500" />

</CardHeader>

<CardContent>

<div className="text-2xl font-bold text-amber-400">{estimates?.length || 0}</div>

</CardContent>

</Card>

</div>

{/\* Financial Management Section \*/}

<div className="grid grid-cols-1 lg:grid-cols-2 gap-6 mb-8">

{/\* Estimates \*/}

<Card className="bg-gray-900 border-amber-500/20">

<CardHeader>

<CardTitle className="text-amber-400">Estimates</CardTitle>

</CardHeader>

<CardContent>

<div className="space-y-4">

{estimates && estimates.length > 0 ? (

estimates.map((estimate: any) => {

const client = clients?.find((c: any) => c.id === estimate.clientId);

return (

<div

key={estimate.id}

className="flex items-center justify-between p-4 bg-gray-800 rounded-lg border border-amber-500/10"

>

<div className="flex-1">

<div className="flex items-center gap-2 mb-1">

<Badge variant="outline" className="text-blue-400 border-blue-400">

EST

</Badge>

<span className="font-medium text-white">{client?.name}</span>

</div>

<div className="text-sm text-gray-400">

{estimate.dogNames} • {estimate.services}

</div>

<div className="text-sm text-gray-400">

{estimate.checkInDate} to {estimate.checkOutDate} ({estimate.totalDays} days)

</div>

<div className="text-lg font-bold text-amber-400">

£{estimate.totalCost}

</div>

</div>

<div className="flex items-center gap-2">

<Button

variant="outline"

size="sm"

className="border-red-500 text-red-500 hover:bg-red-500 hover:text-white"

onClick={() => handleCancelEstimate(estimate.id, estimate.bookingId)}

disabled={deleteEstimateMutation.isPending || deleteBookingMutation.isPending}

>

<Trash2 className="h-4 w-4 mr-1" />

Cancel

</Button>

</div>

</div>

);

})

) : (

<div className="text-center py-8 text-gray-400">

No estimates found

</div>

)}

</div>

</CardContent>

</Card>

{/\* Invoices \*/}

<Card className="bg-gray-900 border-amber-500/20">

<CardHeader>

<CardTitle className="text-amber-400">Invoices</CardTitle>

</CardHeader>

<CardContent>

<div className="space-y-4">

{invoices && invoices.length > 0 ? (

invoices.map((invoice: any) => {

const client = clients?.find((c: any) => c.id === invoice.clientId);

return (

<div

key={invoice.id}

className="flex items-center justify-between p-4 bg-gray-800 rounded-lg border border-amber-500/10"

>

<div className="flex-1">

<div className="flex items-center gap-2 mb-1">

<Badge variant="outline" className="text-green-400 border-green-400">

INV

</Badge>

<span className="font-medium text-white">{client?.name}</span>

</div>

<div className="text-sm text-gray-400">{invoice.description}</div>

<div className="text-lg font-bold text-amber-400">

£{invoice.amount}

</div>

</div>

<Badge

variant="outline"

className={cn(

"ml-2",

invoice.status === "paid"

? "text-green-400 border-green-400"

: "text-yellow-400 border-yellow-400"

)}

>

{invoice.status}

</Badge>

</div>

);

})

) : (

<div className="text-center py-8 text-gray-400">

No invoices found

</div>

)}

</div>

</CardContent>

</Card>

</div>

</div>

</div>

);

}

```

## Problem Summary

The issue is that when you click the red "Cancel" button next to an estimate:

1. ✅ The frontend makes a DELETE request to `/api/estimates/1`

2. ✅ The server responds with 200 status

3. ❌ BUT the server returns HTML (the frontend page) instead of executing the delete route

4. ❌ The estimate data never gets removed from storage

5. ❌ The booking associated with the estimate also doesn't get deleted

The root cause appears to be that the DELETE routes aren't properly registered in the Express server, so requests fall through to serving the frontend instead of executing the deletion logic.

You can provide this complete code to ChatGPT to help resolve the routing issue that's preventing the cancellation system from working properly.