import { and, eq, gt, gte, lt, lte, ne, notInArray, or } from 'drizzle-orm';

import { toZonedTime, format as tzFormat } from 'date-fns-tz';

import { getDay, parseISO, addDays, format, addMinutes, isEqual } from 'date-fns';

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

import type { IStorage } from '../../storage';

import { schedules, docks, appointmentTypes, organizationFacilities } from '@shared/schema';

// Use your actual Drizzle instance type if available

type DrizzleDBInstance = PostgresJsDatabase<typeof import("@shared/schema")>;

export interface AvailabilitySlot {

time: string;

available: boolean;

remainingCapacity: number;

remaining: number;

reason: string;

}

export interface AvailabilityOptions {

testAppointments?: { id: number; startTime: Date; endTime: Date; }[];

}

export async function fetchRelevantAppointmentsForDay(

db: DrizzleDBInstance,

facilityId: number,

dayStart: Date, // Start of day in facility TZ (represented as UTC Date obj)

dayEnd: Date, // Start of NEXT day in facility TZ (represented as UTC Date obj)

effectiveTenantId: number

): Promise<{ id: number; startTime: Date; endTime: Date; }[]> {

console.log(`[fetchRelevantAppointmentsForDay] Fetching for facility ${facilityId}, tenant ${effectiveTenantId}, between ${dayStart.toISOString()} and ${dayEnd.toISOString()}`);

try {

// Ensure db object has the expected methods before chaining

if (!db || typeof db.select !== 'function') {

console.error('[fetchRelevantAppointmentsForDay] Invalid or incomplete DB object passed:', db);

// Throw error so calculateAvailabilitySlots can handle it

throw new Error('Invalid database connection object provided.');

}

const query = db

.select({

id: schedules.id,

startTime: schedules.startTime,

endTime: schedules.endTime,

})

.from(schedules)

.leftJoin(docks, eq(schedules.dockId, docks.id))

.leftJoin(appointmentTypes, eq(schedules.appointmentTypeId, appointmentTypes.id))

.leftJoin(organizationFacilities, eq(docks.facilityId, organizationFacilities.facilityId))

.where(

and(

ne(schedules.dockId, null),

eq(docks.facilityId, facilityId),

// Corrected Overlap Check: (StartA < EndB) and (EndA > StartB)

lt(schedules.startTime, dayEnd), // Appointment starts BEFORE the day ends

gt(schedules.endTime, dayStart), // Appointment ends AFTER day starts

notInArray(schedules.status, ['cancelled', 'rejected']), // Exclude inactive

// Tenant Isolation

or(

eq(organizationFacilities.organizationId, effectiveTenantId),

eq(appointmentTypes.tenantId, effectiveTenantId)

)

)

);

// Execute the query (assuming the chain is awaitable or has .execute())

const relevantSchedules = await query; // Or await query.execute(); depending on Drizzle version/driver

console.log(`[fetchRelevantAppointmentsForDay] Found ${relevantSchedules.length} relevant appointments.`);

// Ensure the return is always an array, even if the query somehow returns non-array

return Array.isArray(relevantSchedules) ? relevantSchedules : [];

} catch (error) {

console.error(`[fetchRelevantAppointmentsForDay] Error fetching appointments:`, error);

// Re-throw a specific error for the caller to handle

throw new Error("Failed to fetch existing appointments.");

}

}

export async function calculateAvailabilitySlots(

db: DrizzleDBInstance,

storage: IStorage,

date: string, // YYYY-MM-DD

facilityId: number,

appointmentTypeId: number,

effectiveTenantId: number,

options?: AvailabilityOptions // Optional parameter for testing

): Promise<AvailabilitySlot[]> {

console.log(`[AvailabilityService] Starting calculation for date=${date}, facilityId=${facilityId}, appointmentTypeId=${appointmentTypeId}, tenantId=${effectiveTenantId}`);

const facility = await storage.getFacility(facilityId, effectiveTenantId);

// \*\* FIXED: Throw immediately if facility check fails \*\*

if (!facility) { throw new Error('Facility not found or access denied.'); }

console.log(`[AvailabilityService] Facility found: ${facility.name}, timezone: ${facility.timezone}`);

const appointmentType = await storage.getAppointmentType(appointmentTypeId);

// \*\* FIXED: Throw immediately if type check fails \*\*

if (!appointmentType) { throw new Error('Appointment type not found or access denied.'); }

// \*\* FIXED: Explicit tenant check after confirming type exists \*\*

if (appointmentType.tenantId && appointmentType.tenantId !== effectiveTenantId) {

console.log(`[AvailabilityService] Tenant mismatch: appointment type ${appointmentTypeId} belongs to tenant ${appointmentType.tenantId}, but request is for tenant ${effectiveTenantId}`);

throw new Error('Appointment type not found or access denied.');

}

const facilityTimezone = facility.timezone || 'America/New\_York';

const zonedDate = toZonedTime(parseISO(`${date}T00:00:00`), facilityTimezone);

const dayOfWeek = getDay(zonedDate); // 0=Sun

console.log(`[AvailabilityService] Date ${date} in ${facilityTimezone} is day of week: ${dayOfWeek}`);

const getObjectField = (obj: any, camelCase: string, snakeCase: string, defaultValue: any = undefined): any => obj?.[snakeCase] ?? obj?.[camelCase] ?? defaultValue;

const getFacilityField = (camelCase: string, snakeCase: string, defaultValue: any = undefined): any => getObjectField(facility, camelCase, snakeCase, defaultValue);

const getAppTypeField = (camelCase: string, snakeCase: string, defaultValue: any = undefined): any => getObjectField(appointmentType, camelCase, snakeCase, defaultValue);

const overrideFacilityHours = getAppTypeField('overrideFacilityHours', 'override\_facility\_hours', false);

const allowAppointmentsThroughBreaks = getAppTypeField('allowAppointmentsThroughBreaks', 'allow\_appointments\_through\_breaks', false);

const appointmentTypeDuration = getAppTypeField('duration', 'duration', 60);

const appointmentTypeBufferTime = getAppTypeField('bufferTime', 'buffer\_time', 0);

const maxConcurrent = getAppTypeField('maxConcurrent', 'max\_concurrent', 1);

console.log(`[AvailabilityService] Settings: overrideHours=${overrideFacilityHours}, allowThroughBreaks=${allowAppointmentsThroughBreaks}, duration=${appointmentTypeDuration}, bufferTime=${appointmentTypeBufferTime}, maxConcurrent=${maxConcurrent}`);

let operatingStartTimeStr = "09:00";

let operatingEndTimeStr = "17:00";

let isOpen = false;

let breakStartTimeStr = "";

let breakEndTimeStr = "";

if (overrideFacilityHours) {

operatingStartTimeStr = "00:00";

operatingEndTimeStr = "23:59"; // Use 23:59 to loop correctly up to the end of the day

isOpen = true;

} else {

const dayKeys = ['sunday', 'monday', 'tuesday', 'wednesday', 'thursday', 'friday', 'saturday'];

const dayKey = dayKeys[dayOfWeek];

isOpen = getFacilityField(`${dayKey}Open`, `${dayKey}\_open`) === true;

if (isOpen) {

operatingStartTimeStr = getFacilityField(`${dayKey}Start`, `${dayKey}\_start`) || "09:00";

operatingEndTimeStr = getFacilityField(`${dayKey}End`, `${dayKey}\_end`) || "17:00";

breakStartTimeStr = getFacilityField(`${dayKey}BreakStart`, `${dayKey}\_break\_start`, "");

breakEndTimeStr = getFacilityField(`${dayKey}BreakEnd`, `${dayKey}\_break\_end`, "");

}

}

if (!isOpen) {

console.log(`[AvailabilityService] Facility ${facility.name} closed on ${date} (DoW: ${dayOfWeek})`);

return [];

}

console.log(`[AvailabilityService] Facility ${facility.name} open on ${date} (${dayOfWeek}) ${operatingStartTimeStr} - ${operatingEndTimeStr}`);

// Calculate date boundaries for fetching appointments

const dayStart = toZonedTime(parseISO(`${date}T00:00:00`), facilityTimezone);

const nextDateStr = format(addDays(parseISO(date), 1), 'yyyy-MM-dd');

const dayEnd = toZonedTime(parseISO(`${nextDateStr}T00:00:00`), facilityTimezone);

// \*\* FIXED: Ensure fetch is only called when options.testAppointments is null or undefined \*\*

let existingAppointments: { id: number; startTime: Date; endTime: Date; }[] = [];

try {

if (options?.testAppointments == null) { // Use == null to check for undefined AND null

existingAppointments = await fetchRelevantAppointmentsForDay(db, facilityId, dayStart, dayEnd, effectiveTenantId);

} else {

existingAppointments = options.testAppointments;

console.log(`[AvailabilityService] Using ${existingAppointments.length} test appointments`);

}

} catch (fetchError) {

console.error("[AvailabilityService] Error during fetchRelevantAppointmentsForDay:", fetchError);

// Propagate the specific error from fetchRelevantAppointmentsForDay

throw fetchError;

}

const result: AvailabilitySlot[] = [];

const slotIntervalMinutes = Math.max(appointmentTypeBufferTime > 0 ? appointmentTypeBufferTime : appointmentTypeDuration, 15);

// Create Date objects for operating start/end IN THE FACILITY'S TIMEZONE

const operatingStartDateTime = toZonedTime(parseISO(`${date}T${operatingStartTimeStr}`), facilityTimezone);

let operatingEndDateTime = toZonedTime(parseISO(`${date}T${operatingEndTimeStr}`), facilityTimezone);

// Adjust end time for loop comparison

// If 23:59, treat as END of the day (start of next day)

if (operatingEndTimeStr === "23:59") {

operatingEndDateTime = dayEnd;

} else if (operatingEndDateTime <= operatingStartDateTime) {

operatingEndDateTime = addDays(operatingEndDateTime, 1);

}

let breakStartDateTime: Date | null = null;

let breakEndDateTime: Date | null = null;

if (breakStartTimeStr && breakEndTimeStr && breakStartTimeStr.includes(':') && breakEndTimeStr.includes(':')) {

try {

// \*\* FIXED: Use dayStart (which is zoned) as base for break times \*\*

breakStartDateTime = new Date(dayStart);

breakStartDateTime.setHours(parseInt(breakStartTimeStr.split(':')[0], 10), parseInt(breakStartTimeStr.split(':')[1], 10), 0, 0);

breakEndDateTime = new Date(dayStart);

breakEndDateTime.setHours(parseInt(breakEndTimeStr.split(':')[0], 10), parseInt(breakEndTimeStr.split(':')[1], 10), 0, 0);

if (breakEndDateTime <= breakStartDateTime) { breakEndDateTime = addDays(breakEndDateTime, 1); }

console.log(`[AvailabilityService] Break time for ${date} (Local): ${tzFormat(breakStartDateTime, 'yyyy-MM-dd HH:mm:ss zzzz', { timeZone: facilityTimezone })} to ${tzFormat(breakEndDateTime, 'yyyy-MM-dd HH:mm:ss zzzz', { timeZone: facilityTimezone })}`);

} catch (e) { console.error("Error parsing break times", e); breakStartDateTime = null; breakEndDateTime = null; }

}

let currentSlotStartTime = new Date(operatingStartDateTime);

while (currentSlotStartTime < operatingEndDateTime) {

const currentSlotEndTime = addMinutes(currentSlotStartTime, appointmentTypeDuration);

// \*\* FIXED: Loop Termination Check - Allow slots ENDING AT end time \*\*

if (currentSlotEndTime > operatingEndDateTime) {

console.log(`[AvailabilityService] Slot starting at ${tzFormat(currentSlotStartTime, 'HH:mm', { timeZone: facilityTimezone })} duration ${appointmentTypeDuration}m ends after operating end ${operatingEndTimeStr}. Stopping.`);

break;

}

let isSlotAvailable = true;

let reason = "";

let conflictingApptsCount = 0;

// Check for conflicts

if (existingAppointments && existingAppointments.length > 0) {

conflictingApptsCount = existingAppointments.filter((appt) => {

const apptStart = appt.startTime.getTime(); // Compare epoch ms (UTC)

const apptEnd = appt.endTime.getTime();

const slotStart = currentSlotStartTime.getTime();

const slotEnd = currentSlotEndTime.getTime();

// \*\* FIXED: Correct Overlap Logic: (StartA < EndB) && (EndA > StartB) \*\*

return apptStart < slotEnd && apptEnd > slotStart;

}).length;

}

// \*\* FIXED: Check Capacity FIRST \*\*

const currentCapacity = maxConcurrent - conflictingApptsCount;

if (currentCapacity <= 0) {

isSlotAvailable = false;

reason = "Capacity full";

}

// Check break time ONLY IF slot is still potentially available based on capacity

if (isSlotAvailable && breakStartDateTime && breakEndDateTime) {

// \*\* FIXED: Compare slot interval with break interval \*\*

if (currentSlotStartTime.getTime() < breakEndDateTime.getTime() && currentSlotEndTime.getTime() > breakStartDateTime.getTime()) {

if (!allowAppointmentsThroughBreaks) {

isSlotAvailable = false;

reason = "Break Time"; // Override reason

} else {

// Only add note if not already marked unavailable by capacity

if (isSlotAvailable) {

reason = "Spans through break time";

}

console.log(`[AvailabilityService] Slot ${tzFormat(currentSlotStartTime, 'HH:mm', { timeZone: facilityTimezone })} spans break time, but allowed.`);

}

}

}

const remainingCapacity = isSlotAvailable ? Math.max(0, currentCapacity) : 0;

// Final status check: If capacity is zero, ensure unavailable

if (remainingCapacity <= 0) {

isSlotAvailable = false;

// Set reason to Capacity Full only if it wasn't already set to Break Time

if (reason !== "Break Time") {

reason = "Capacity full";

}

}

result.push({

time: tzFormat(currentSlotStartTime, 'HH:mm', { timeZone: facilityTimezone }),

available: isSlotAvailable,

remainingCapacity: remainingCapacity,

remaining: remainingCapacity,

// \*\* FIXED: Corrected reason logic \*\*

reason: isSlotAvailable ? (reason === "Spans through break time" ? reason : "") : reason,

});

currentSlotStartTime = addMinutes(currentSlotStartTime, slotIntervalMinutes);

}

return result;

}