# Exercise.no:17 Proximity Alerter Web Application

## Tools Used :

* **Framework**: React with TypeScript
* **UI Components**: ShadCN/UI with Tailwind CSS
* **State Management**: React Context API
* **Local Storage**: Browser’s localStorage APIb
* **Geolocation**: Browser’s Geolocation API
* **Audio**: Web Audio API
* **Vibration**: Vibration API

## Problem Statement :

Build a web application that alerts users when they move more than 10 meters away from a set target location. The application should: 1. Allow users to set a target location (current location or custom coordinates) 2. Track the user’s current location 3. Calculate the distance between current and target locations 4. Trigger alerts (sound, vibration) when the distance exceeds the threshold 5. Provide settings to customize alert preferences 6. Work in browsers with or without geolocation permissions

## Code :

### 1. Data Schema (shared/schema.ts) :

// Core data types for the application  
export const locationSchema = z.object({  
 latitude: z.number(),  
 longitude: z.number(),  
 timestamp: z.number(),  
});  
  
export const settingsSchema = z.object({  
 alertDistance: z.number().default(10), // Default 10 meters  
 soundEnabled: z.boolean().default(true),  
 vibrationEnabled: z.boolean().default(true),  
});  
  
export const targetLocationSchema = z.object({  
 location: locationSchema,  
 label: z.string().optional(),  
 createdAt: z.number(),  
});  
  
export type Location = z.infer<typeof locationSchema>;  
export type Settings = z.infer<typeof settingsSchema>;  
export type TargetLocation = z.infer<typeof targetLocationSchema>;

### 2. Location Context Provider (client/src/hooks/useLocation.tsx) :

// Core state management hook for the application  
export const LocationProvider: React.FC<{children: React.ReactNode}> = ({ children }) => {  
 // State variables  
 const [currentLocation, setCurrentLocation] = useState<Location | null>(null);  
 const [targetLocation, setTargetLocationState] = useState<TargetLocation | null>(null);  
 const [distance, setDistance] = useState<number | null>(null);  
 const [isTracking, setIsTracking] = useState(false);  
 const [isExceedingLimit, setIsExceedingLimit] = useState(false);  
 const [isLoading, setIsLoading] = useState(true);  
 const [settings, setSettings] = useState<Settings>(DEFAULT\_SETTINGS);  
   
 // Watch position ID reference  
 const watchId = useRef<number | null>(null);  
 // Audio element reference  
 const audioRef = useRef<HTMLAudioElement | null>(null);  
   
 // Toast notifications  
 const { toast } = useToast();  
  
 // Initialize location services and load stored data  
 useEffect(() => {  
 // Implementation details...  
 }, []);  
  
 // Calculate distance when location changes  
 useEffect(() => {  
 // Implementation details...  
 }, [currentLocation, targetLocation, settings.alertDistance]);  
  
 // Core functions  
 const startLocationTracking = async () => {  
 // Handle geolocation or fallback to mock location if unavailable  
 if (!('geolocation' in navigator)) {  
 // Fallback handling  
 setCurrentLocation({  
 latitude: 40.7128, // Example coordinates  
 longitude: -74.006,  
 timestamp: Date.now(),  
 });  
 setIsTracking(true);  
 return;  
 }  
   
 // Start watching position  
 // Implementation details...  
 };  
  
 // Public API  
 return (  
 <LocationContext.Provider  
 value={{  
 currentLocation,  
 targetLocation,  
 distance,  
 isTracking,  
 settings,  
 isExceedingLimit,  
 isLoading,  
 setTargetLocation,  
 startTracking,  
 stopTracking,  
 updateSettings,  
 }}  
 >  
 {children}  
 </LocationContext.Provider>  
 );  
};  
  
// Hook to use location context  
export const useLocation = () => useContext(LocationContext);

### **3. Target Location Form (client/src/components/TargetLocationForm.tsx) :**

export function TargetLocationForm() {  
 const { currentLocation, targetLocation, setTargetLocation, isLoading } = useLocation();  
 const [label, setLabel] = useState('');  
 const [isSubmitting, setIsSubmitting] = useState(false);  
 const [manualLat, setManualLat] = useState<string>('');  
 const [manualLng, setManualLng] = useState<string>('');  
 const [useCurrentLocation, setUseCurrentLocation] = useState(true);  
   
 const handleSetTargetLocation = async () => {  
 setIsSubmitting(true);  
   
 try {  
 let latitude: number;  
 let longitude: number;  
   
 if (useCurrentLocation) {  
 // Use current location  
 if (!currentLocation) {  
 alert('Unable to set target location. Please wait for your current location or enter coordinates manually.');  
 setIsSubmitting(false);  
 return;  
 }  
 latitude = currentLocation.latitude;  
 longitude = currentLocation.longitude;  
 } else {  
 // Use manual coordinates  
 // Input validation...  
 latitude = parseFloat(manualLat);  
 longitude = parseFloat(manualLng);  
 }  
   
 const newTargetLocation: TargetLocation = {  
 location: {  
 latitude,  
 longitude,  
 timestamp: Date.now(),  
 },  
 label: label.trim() || undefined,  
 createdAt: Date.now(),  
 };  
   
 await setTargetLocation(newTargetLocation);  
 // Success handling...  
 } catch (error) {  
 // Error handling...  
 } finally {  
 setIsSubmitting(false);  
 }  
 };  
   
 // Render form with toggle between current and manual location input  
 return (  
 <div className="bg-white p-4 rounded-lg shadow mb-4">  
 <h3 className="text-md font-bold text-gray-700 mb-3">Set Target Location</h3>  
   
 {/\* Toggle between current and manual location \*/}  
 <div className="flex items-center mb-4">  
 {/\* Toggle UI \*/}  
 </div>  
   
 {/\* Manual coordinate inputs when not using current location \*/}  
 {!useCurrentLocation && (  
 <div className="space-y-3 mb-4">  
 {/\* Latitude input \*/}  
 {/\* Longitude input \*/}  
 </div>  
 )}  
   
 {/\* Location label input \*/}  
 <div className="mb-4">  
 {/\* Label input \*/}  
 </div>  
   
 {/\* Action buttons \*/}  
 <div className="flex gap-2">  
 {/\* Set/Update target button \*/}  
 {/\* Clear target button (when target exists) \*/}  
 </div>  
 </div>  
 );  
}

### 4. Distance Display and Alert (client/src/components/DistanceAlert.tsx) :

export function DistanceAlert() {  
 const { distance, isExceedingLimit, settings } = useLocation();  
   
 if (distance === null) {  
 return null;  
 }  
   
 return (  
 <div className={`p-4 rounded-lg shadow mb-4 ${  
 isExceedingLimit ? 'bg-red-500 text-white animate-pulse' : 'bg-green-500 text-white'  
 }`}>  
 <div className="flex justify-between items-center">  
 <div>  
 <h3 className="font-bold">  
 {isExceedingLimit ? 'ALERT: Distance Exceeded!' : 'Within Safe Range'}  
 </h3>  
 <p>  
 Current distance: <strong>{formatDistance(distance)}</strong>  
 </p>  
 <p>  
 Maximum allowed: <strong>{formatDistance(settings.alertDistance)}</strong>  
 </p>  
 </div>  
 <div className={`text-4xl ${isExceedingLimit ? 'text-white' : 'text-green-200'}`}>  
 {isExceedingLimit ? '⚠️' : '✓'}  
 </div>  
 </div>  
 </div>  
 );  
}

### 5. Distance Calculation (client/src/lib/utils.ts) :

/\*\*  
 \* Calculate the distance between two points using the Haversine formula  
 \*/  
export function calculateDistance(  
 lat1: number,  
 lon1: number,  
 lat2: number,  
 lon2: number  
): number {  
 // Convert latitude and longitude from degrees to radians  
 const radLat1 = (lat1 \* Math.PI) / 180;  
 const radLon1 = (lon1 \* Math.PI) / 180;  
 const radLat2 = (lat2 \* Math.PI) / 180;  
 const radLon2 = (lon2 \* Math.PI) / 180;  
  
 // Haversine formula  
 const dLat = radLat2 - radLat1;  
 const dLon = radLon2 - radLon1;  
 const a =  
 Math.sin(dLat / 2) \* Math.sin(dLat / 2) +  
 Math.cos(radLat1) \* Math.cos(radLat2) \*  
 Math.sin(dLon / 2) \* Math.sin(dLon / 2);  
 const c = 2 \* Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));  
   
 // Earth's radius in meters  
 const R = 6371000;  
   
 // Distance in meters  
 return R \* c;  
}

## Output :

