

# Abschlusspräsentation GPS-Tracking App

# Inhaltsverzeichnis

- Aufgabenstellung
- Anwendungsfälle
- Zieldefinition
- Use Cases / Anforderungsanalyse
- Requirement Diagramm
- Datenbankmodell
- Skizzen / Umsetzung
- Wichtige Code Blöcke
- Live Demonstration in der App

# Aufgabenstellung

- Schnittstelle zur Erfassung und Darstellung von Bewegungsdaten
- Streckenanzeige auf einer Karte
- Distanz, Dauer, Höhenmeter und Geschwindigkeiten sollen erfasst werden
- Die Aufzeichnung der Strecke soll steuerbar und speicherbar sein
- Aufzeichnungen sollen im Nachhinein aufrufbar/einsehbar sein
- Es soll Gesamtstatistiken aus allen Aufzeichnungen geben

# Anwendungsfälle

- Der Nutzer soll sich mit der Karte orientieren können und seine Route planen.
- Der Nutzer soll die Strecke, welche er zurücklegt, auf der Karte verfolgen und aufzeichnen.
- Der Nutzer soll persönliche Ziele in Form von Geschwindigkeit oder Distanz setzen und mit Hilfe einer Live-Anzeige nachverfolgen .
- Der Nutzer soll seine Routen auch später noch ansehen, um Fortschritte zu erkennen oder seine Leistungen vergleichen zu können.
- Der Nutzer soll sich die Zusammenfassung aller Einzelrouten ansehen, um zu sehen, wie viel er schon "geleistet" hat.
- Der Nutzer kann Fotos aufnehmen, um gewisse Punkte auf der Strecke festzuhalten. Diese kann er sich später auch angucken, und sehen wo diese Aufgenommen wurden.
- Der Nutzer soll die App individuell anpassen können.
  - Kartendesign ändern
  - Hell- oder Dunkelmodus einschalten
  - Genauigkeit des GPS Signals einstellen um selbst Ressourcenbedarf zu bestimmen (Mobile Daten/Speicher)

# Zieldefinition

- Entwicklung einer mobilen Android-Applikation zur Aufzeichnung von Outdoor-Aktivitäten wie Fahrradfahren oder Spazierengehen.
- Die App soll den zurückgelegten Weg visuell darstellen, relevante Aktivitätsdaten wie Distanz und Dauer darstellen.
- Diese sollen aufgezeichnet und lokal gespeichert werden
- Nutzer sollen ihre Daten einsehen und (optional) statistische Auswertungen vornehmen können.

# Anforderungsanalyse



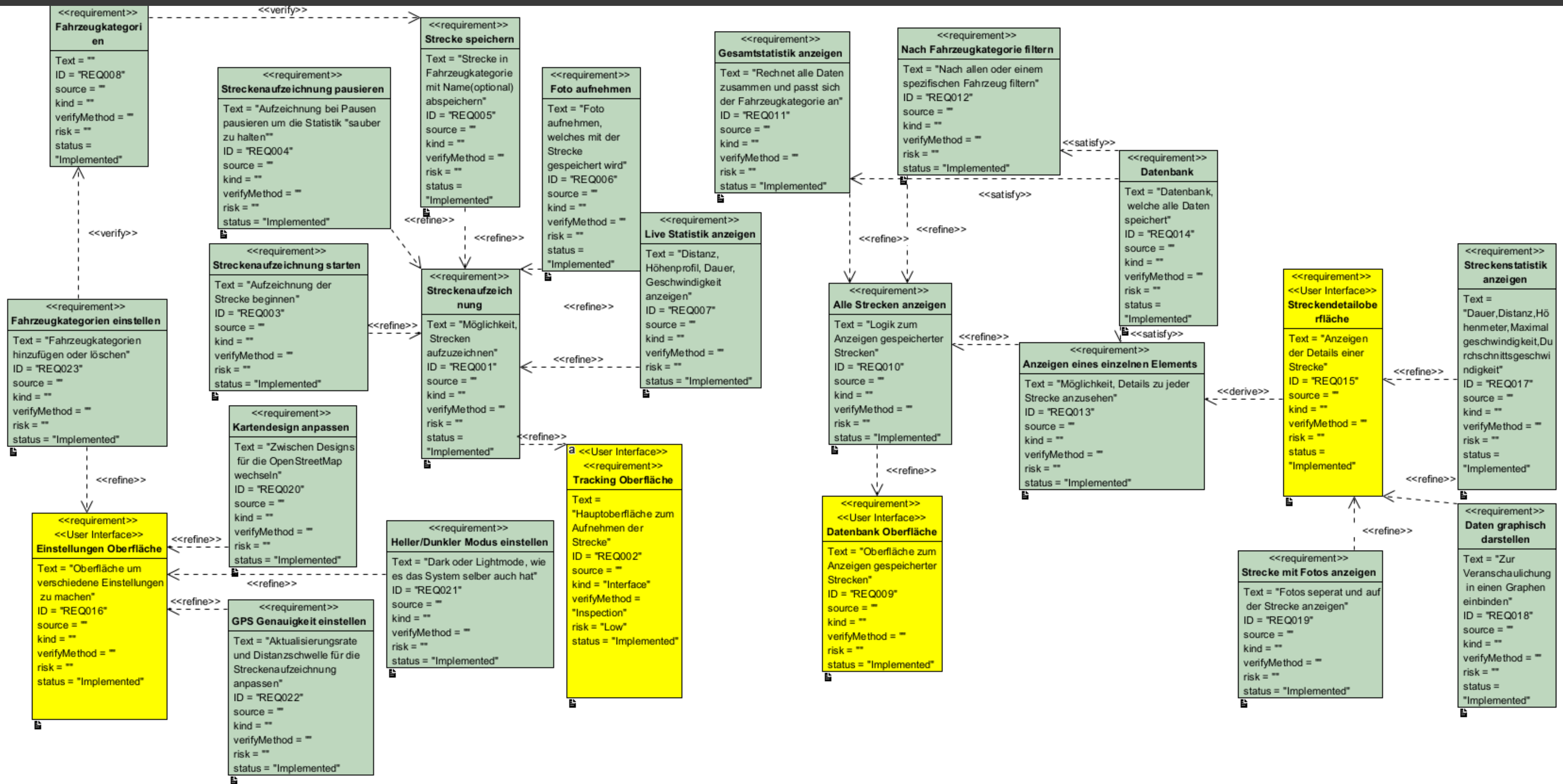
## Funktionale Anforderungen:

- Erfassung und Anzeige von Bewegungsdaten
- Messung relevanter Aktivitätsdaten
- Steuerbare Aufzeichnung
- Lokale Speicherung
- Abruf vergangener Strecken
- Statistische Auswertungen

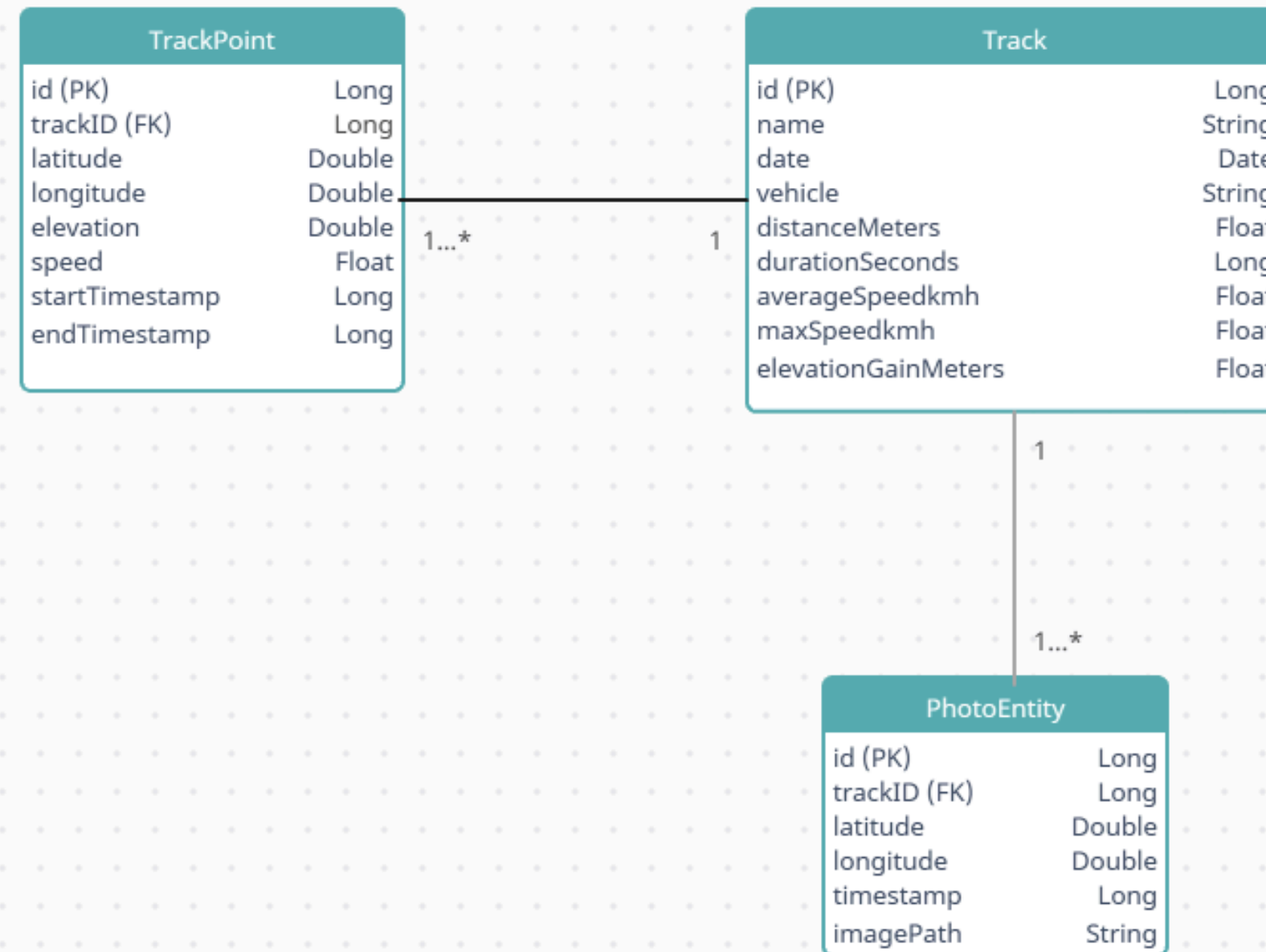
## Nicht-funktionale Anforderungen:

- Mobile Android-App
- Performante und ressourcenschonende Umsetzung
- "Datenschutz" durch lokale Speicherung
- Einfach Bedienung/ Benutzerfreundlichkeit
- Play-Store fähig

# Requirement Diagram

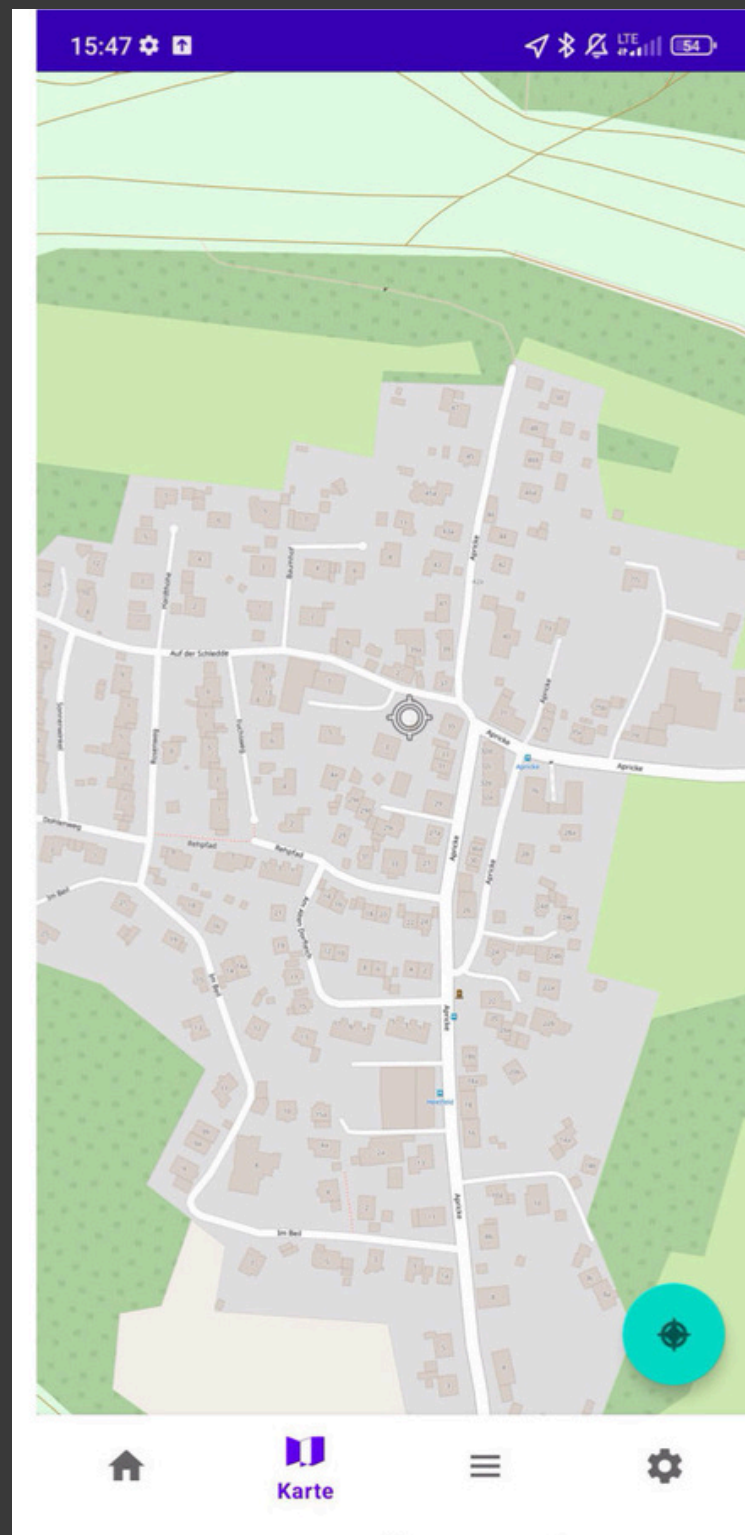


# Datenbankmodell

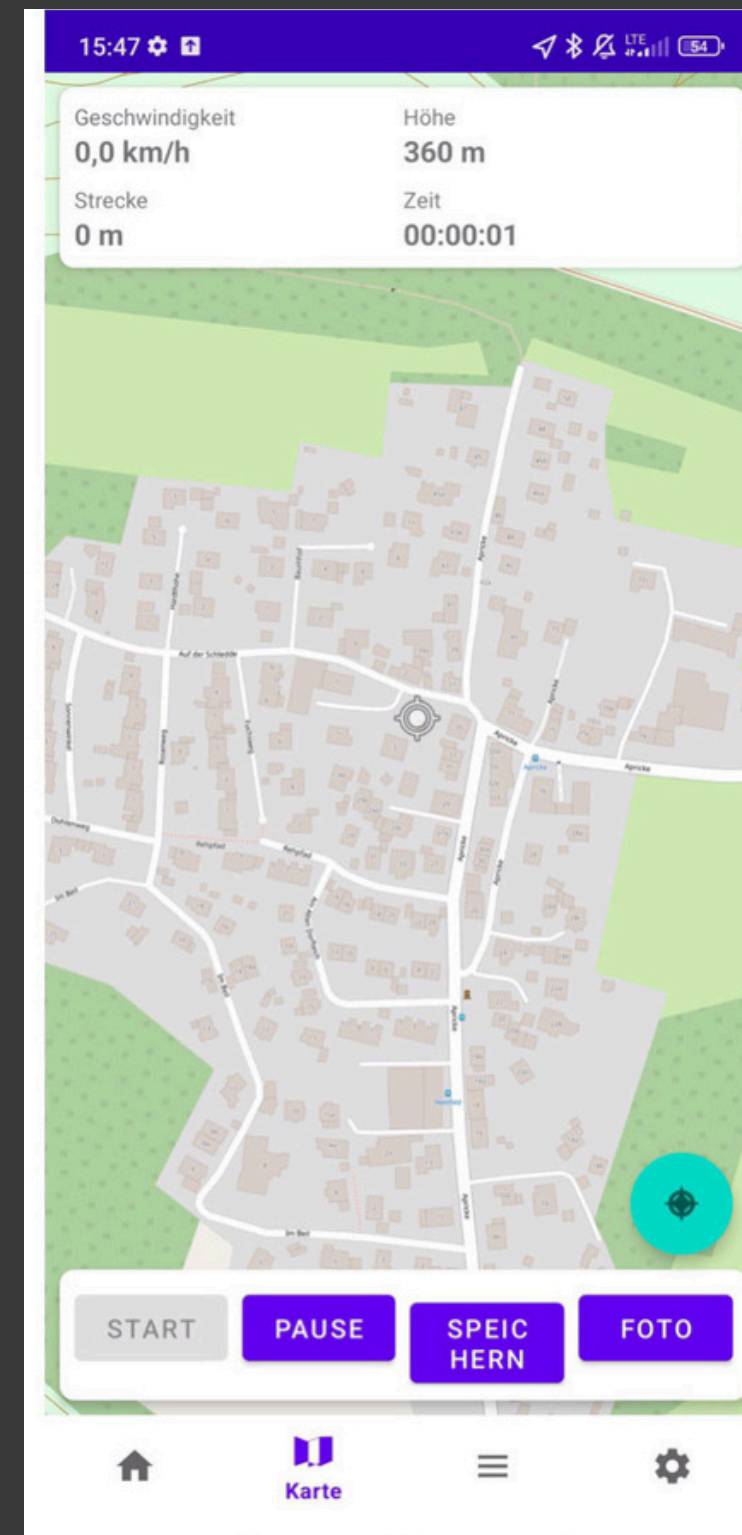
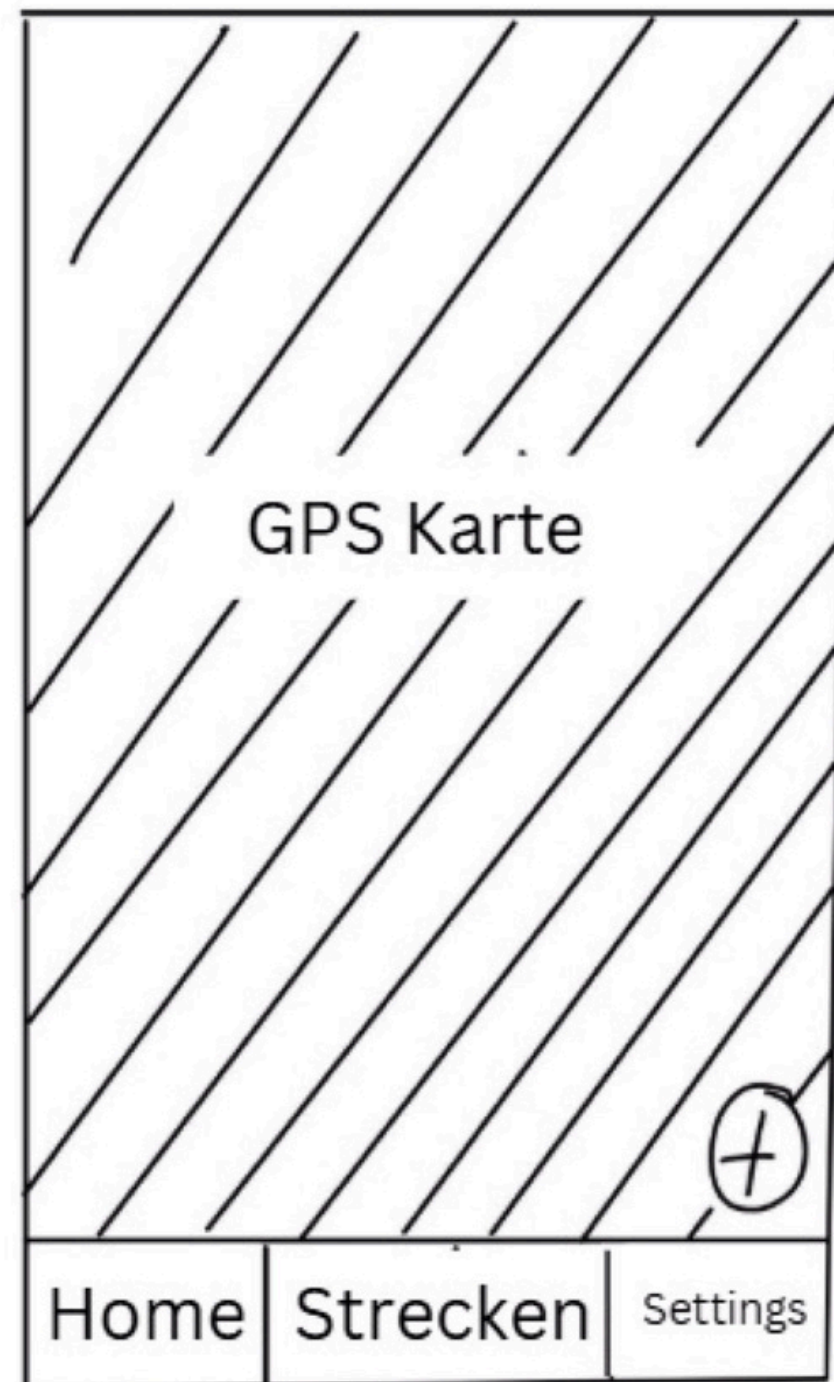




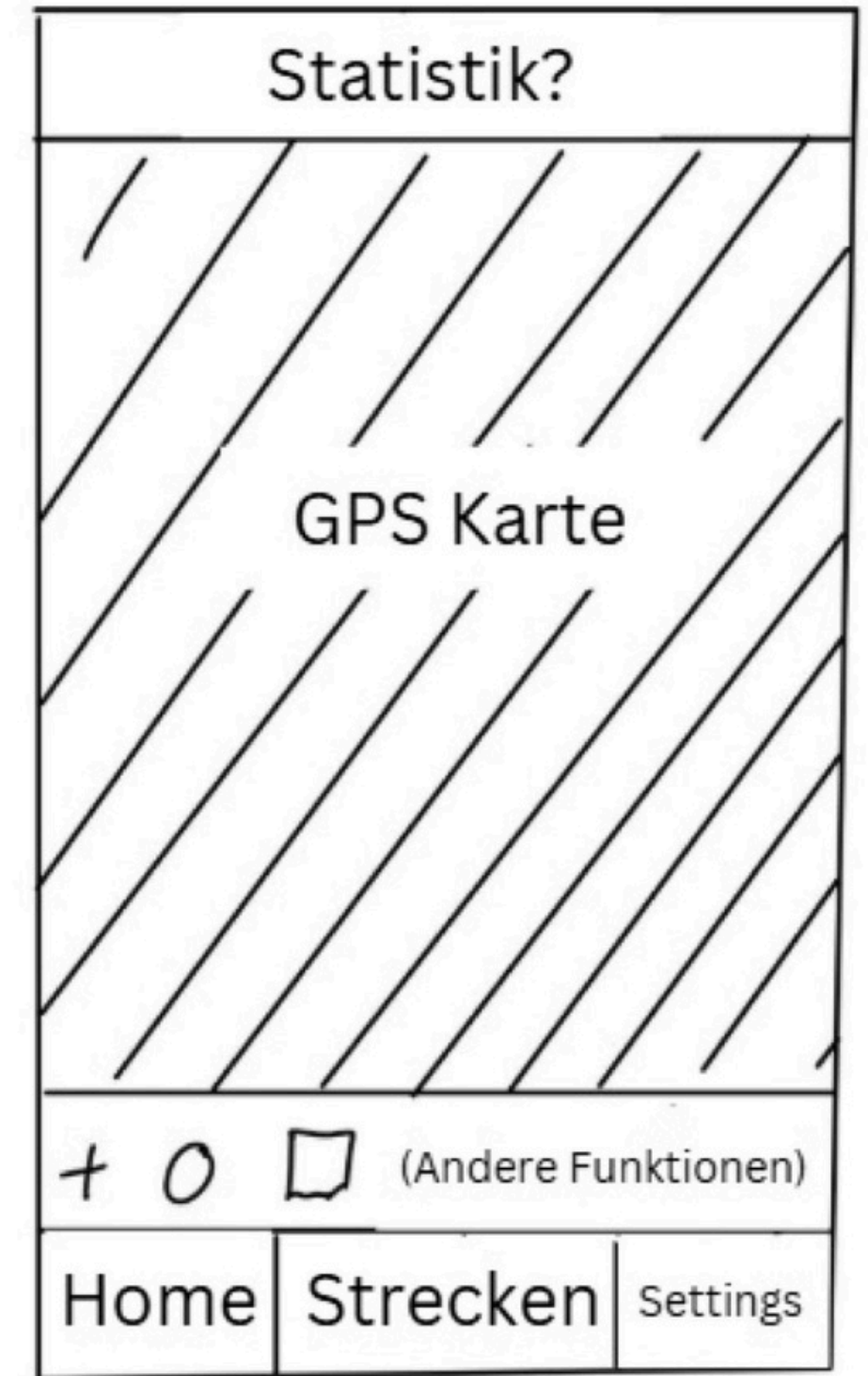
# Skizze 1: Hauptbildschirm



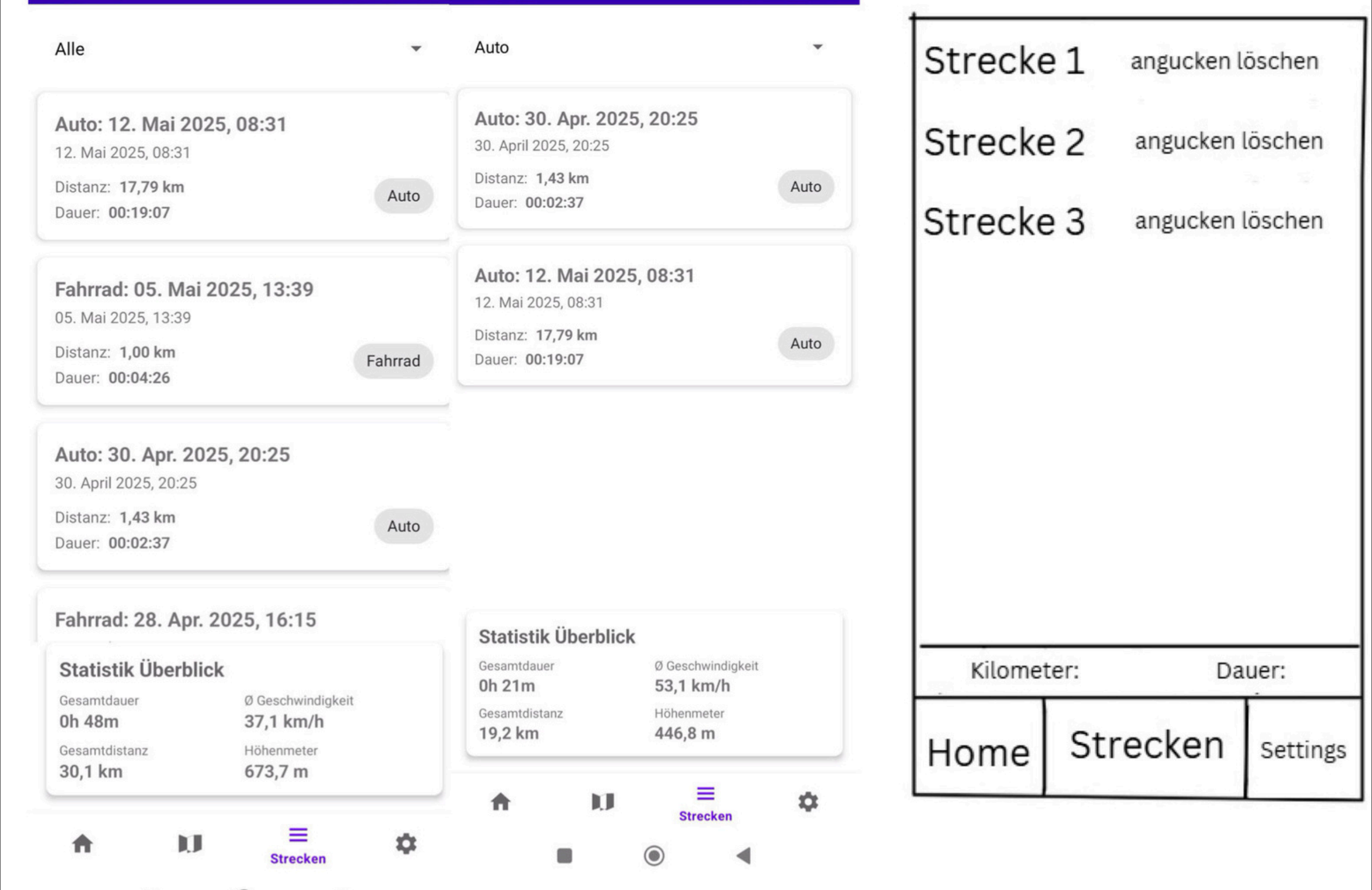
Ohne Tracking



Aktives Tracking




# Skizze 2: Strecken Bildschirm





# Wichtige Code Blöcke

```
class LocationTracker(  
  
    /**  
     * Startet den Empfang von Standortaktualisierungen.  
     *  
     * Diese Methode:  
     * 1. Stoppt bestehende Updates, falls vorhanden  
     * 2. Aktualisiert die LocationRequest-Konfiguration  
     * 3. Versucht den letzten bekannten Standort zu erhalten für sofortige Rückmeldung  
     * 4. Startet kontinuierliche Standortaktualisierungen  
     *  
     * Erfordert bereits erteilte Standortberechtigungen.  
     */  
    @SuppressWarnings("MissingPermission")  
    fun startLocationUpdates() {  
        try {  
            // Stoppe zuerst bestehende Updates, um sicherzustellen, dass Einstellungen angewendet werden  
            stopLocationUpdates()  
  
            // Aktualisiere die LocationRequest mit aktuellen Einstellungen  
            locationRequest = createLocationRequest()  
  
            // Hole zuerst den letzten bekannten Standort für sofortige Antwort  
            fusedLocationClient.lastLocation.addOnSuccessListener { location ->  
                location?.let {  
                    Log.d(TAG, "Letzter bekannter Standort als Initialposition verwendet")  
                    onLocationUpdate(it)  
                } ?: Log.d(TAG, "Kein letzter bekannter Standort verfügbar")  
            }  
  
            // Dann starte kontinuierliche Updates  
            fusedLocationClient.requestLocationUpdates(  
                locationRequest,  
                locationCallback,  
                 Looper.getMainLooper() // Haupt-Looper für UI-Updates, läuft auf dem Main Thread  
            )  
            isTrackingStarted = true  
            Log.d(TAG, "Standortaktualisierungen gestartet mit Intervall: ${locationRequest.intervalMillis}ms")  
        } catch (e: Exception) {  
            Log.e(TAG, "Fehler beim Starten der Standortaktualisierungen", e)  
        }  
    }  
}
```

# Wichtige Code Blöcke

Aktualisiert die  
Position auf der  
Karte

```
/**
 * Aktualisiert die Kartenposition und den Positionsmarker.
 * Zentriert die Karte bei der ersten Position oder bei aktivem Tracking.
 *
 * @param geoPoint Die neue Position als GeoPoint
 */
private fun updateMapLocation(geoPoint: GeoPoint) {
    Log.d(TAG, "Updating map location: lat=${geoPoint.latitude}, lng=${geoPoint.longitude}")

    // Karte bei erster Positionsaktualisierung oder aktivem Tracking zentrieren
    if (!hasInitialLocation || viewModel.isTracking.value) {
        Log.d(TAG, "Zentriere Karte auf neue Position")
        mapView.controller.animateTo(geoPoint)
        hasInitialLocation = true
    }

    // Positionsmarker aktualisieren oder erstellen
    if (!locationMarker.isInitialized) {
        Log.d(TAG, "Neuer Positionsmarker wird erstellt")
        locationMarker = Marker(mapView).apply {
            position = geoPoint
            setAnchor(Marker.ANCHOR_CENTER, Marker.ANCHOR_BOTTOM)
            icon = ContextCompat.getDrawable(requireContext(), android.R.drawable.ic_menu_mylocation)
            title = "Aktuelle Position"
            mapView.overlays.add(this)
        }
    } else {
        locationMarker.position = geoPoint
    }

    mapView.invalidate()
}
```

# Wichtige Code Blöcke

Aktualisiert die  
Streckenaufzeichnung  
(Den blauen Weg)

```
/**  
 * Aktualisiert die angezeigte Route auf der Karte.  
 *  
 * @param points Liste von GeoPoints, die die Route bilden  
 */  
private fun updateRouteOnMap(points: List<GeoPoint>) {  
    if (points.isNotEmpty()) {  
        routePolyline.setPoints(points)  
  
        // Sicherstellen, dass die Polyline im Overlay-Layer ist  
        if (!mapView.overlays.contains(routePolyline)) {  
            mapView.overlays.add(routePolyline)  
        }  
    } else {  
        // Polyline leeren, wenn keine Punkte vorhanden  
        routePolyline.setPoints(emptyList())  
    }  
  
    // Karte neu zeichnen  
    mapView.invalidate()  
}
```

# Wichtige Code Blöcke

```
private val trackRepository = TrackRepository(application)
```

```
private val _track = MutableLiveData<Track>()
```

```
val track: LiveData<Track> = _track //Live Data für statische Daten, die nicht oft aktualisiert werden
```

```
private val _trackPoints = MutableStateFlow<List<TrackPoint>>(emptyList())
```

```
val trackPoints: StateFlow<List<TrackPoint>> = _trackPoints //State Flow für Streams/Große Datenmengen die mit collect geladen werden
```

```
private val _photos = MutableStateFlow<List<PhotoEntity>>(emptyList())
```

```
val photos: StateFlow<List<PhotoEntity>> = _photos
```

```
init {
```

```
    loadTrack()
```

```
    loadTrackPoints()
```

```
    loadPhotos()
```

```
}
```

```
/**
```

```
 * Lädt die grundlegenden Informationen des Tracks aus dem Repository
```

```
 */
```

```
private fun loadTrack() {
```

```
    viewModelScope.launch { //Erstelle eine Coroutine, in einen anderen Thread
```

```
        val track = trackRepository.getTrackById(trackId)
```

```
        _track.value = track
```

```
    }
```

```
}
```

**Vielen Dank für ihre Aufmerksamkeit!**