**Overview:**

The project consists of two sections application and geo fence monitor library. Geo fence library is responsible for monitoring all the geo fences and can work independently with any other application.

**Application:**

1. User Flow:

* Open TrackMe app.
* Give location permission.
* Enable GPS (if not enabled already).
* Long click on any location on map to add geo fence to monitor.
* Provide geo fence name and radius in meters.
* See notifications in notification area when user enters or leaves the geo fence.
* Notification history can also be viewed in history area.

1. **Design:**

* Application is built on MVC design pattern and highly scalable.
* Used Realm database for saving geo fence data and notification history.

1. **Technical Flow:**

* When app started it checks for location permission and request if not granted already.
* On permission success app tries to get last known location from these providers:
  + GPS\_PROVIDER
  + NETWORK\_PROVIDER
  + PASSIVE\_PROVIDER
* If no last known location found app tries to get user location.
* When a new geo fence is added application send this geo fence to geo fence monitor library to monitor.
* Add listeners to get updates from library like current power profile, geo fence enter, geo fence exit.

**Geo fence Monitor (library):**

1. **Design:**

* Bounded service is used to monitor geo fences.
* **GeofenceMonitor** interface exposes service calls to application**.**
* **GeofenceMonitor** is an interface so client can monitor geo fences without knowing the actual implementation.
* **GeofenceManager** class returns single instance of **GeofenceMonitor.**
* **GeofenceMonitorImpl** is a wrapper to the bounded service in order to limit direct user interaction with service.
* **GeofenceService** is a bounded service and actual implementation of GeofenceMonitor.
* **MotionDetector** used to detect if there is any significant motion, this class is used when significant motion strategy is enabled.

1. **Technical Flow:**

* Whenever a geo fence is added service starts monitoring it.
* If there is no geo fence to monitor service will turn off all location providers.
* At every one second service tunes the location request strategy based on the following parameters:
  + Distance from user current location to the nearest geo fence entry point.
  + Distance from user current location to the nearest geo fence exit point.
  + If there is significant motion (when significant motion strategy is enabled).
* Based user distance from current location to nearest geo fence entry/exit point following strategies are used in order to request location updates efficiently and save battery power:
* High Power (distanceToNearestGeoFenceEvent < 20):
  + Minimum time between location updates 0.5s
  + Minimum distance between location updates 0
  + Minimum acceptable accuracy 5m
* Med Power (distanceToNearestGeoFenceEvent < 50):
  + Minimum time between location updates 2s
  + Minimum distance between location updates 2m
  + Minimum acceptable accuracy 15m
* Low Power (distanceToNearestGeoFenceEvent < 100):
  + Minimum time between location updates 5s
  + Minimum distance between location updates 3m
  + Minimum acceptable accuracy 30m
* Low Power Network Provider Only (distanceToNearestGeoFenceEvent>100):
  + Minimum time between location updates 5s
  + Minimum distance between location updates 3m
  + Minimum acceptable accuracy 30m
* Idle (If there is no significant motion and SMS is enabled):
  + Disable all providers to save power
  + Minimum distance between location updates 3m
  + Minimum acceptable accuracy 30m

**Conclusion:**

Location request strategies can be fine-tuned based on application specific requirements. This project relies on two constraints for requesting location (Distance to Nearest Geo Fence Event and Significant Motion). There are many other things that can provide valuable information about requesting location updates efficiently like getting data from different sensors, recoding user habits and predict, sensor fusion.