Handle route deviations

Last Updated Aug 29, 2025 11 minute read #HERE SDK #Android #Developer guide #Active

Navigation is only available with the Navigate license.

If you detect a route deviation, you can decide based on `distanceInMeters` if you want to reroute users to their destination. Note that for a full route recalculation you may want to use the same route parameters.

This is how it works:

1. Continuously listen for `RouteDeviation` events.
2. Based on the deviation in meters decide what you want to do next. For example:
   a. Create a new `Route` and use it for navigation instead of the previous `Route`.
   b. Or use `returnToRoute()` to keep using the existing `Route`.
   c. Keep following the old `Route` without a change. Risk: If the deviation is too far, the user might not reach the next maneuver or miss maneuvers.

Getting notified on deviations

See the below code on getting an `RouteDeviation` event:

Java Kotlin

Explain this code

```
// Notifies on a possible deviation from the route.
visualNavigator.routeDeviationListener =
    RouteDeviationListener { routeDeviation: RouteDeviation ->
        val route = visualNavigator.route
            ?: // May happen in rare cases when route was set to null inbetween.
            return@RouteDeviationListener
        // Get current geographic coordinates.
        val currentMapMatchedLocation = routeDeviation.currentLocation.mapMatchedLocation
        val currentGeoCoordinates = currentMapMatchedLocation?.coordinates
            ?: routeDeviation.currentLocation.originalLocation.coordinates

        // Get last geographic coordinates on route.
        val lastGeoCoordinatesOnRoute: GeoCoordinates?
        if (routeDeviation.lastLocationOnRoute != null) {
            val lastMapMatchedLocationOnRoute =
                routeDeviation.lastLocationOnRoute!!.mapMatchedLocation
            lastGeoCoordinatesOnRoute = lastMapMatchedLocationOnRoute?.coordinates
                ?: routeDeviation.lastLocationOnRoute!!.originalLocation.coordinates
        } else {
            Log.d(
                TAG,
                "User was never following the route. So, we take the start of the route instead."
            )
            lastGeoCoordinatesOnRoute = route.sections[0].departurePlace.originalCoordinates
        }

        val distanceInMeters = currentGeoCoordinates.distanceTo(
            lastGeoCoordinatesOnRoute!!
        ).toInt()
        Log.d(
            TAG,
            "RouteDeviation in meters is $distanceInMeters"
        )

        // Now, an application needs to decide if the user has deviated far enough and
        // what should happen next: For example, you can notify the user or simply try to
        // calculate a new route. When you calculate a new route, you can, for example,
        // take the current location as new start and keep the destination - another
        // option could be to calculate a new route back to the lastMapMatchedLocationOnRoute.
        // At least, make sure to not calculate a new route every time you get a RouteDeviation
        // event as the route calculation happens asynchronously and takes also some time to
        // complete.
        // The deviation event is sent any time an off-route location is detected: It may make
        // sense to await around 3 events before deciding on possible actions.
    }
```

In the above example, we calculate the distance based on the coordinates contained in `RouteDeviation`: `distanceInMeters`. This indicates the straight-line distance between the expected location on the route and your actual location. If that is considered too far, you can set a newly calculated route to the `VisualNavigator` instance - and all further events will be based on the new route.

Keep in mind, that in a drive guidance scenario, `lastLocationOnRoute` and `mapMatchedLocation` can be `null`. If `routeDeviation.lastLocationOnRoute` is `null`, then the user was never following the route - this can happen when the starting position is farther away from the road network. Usually, the `Navigator` / `VisualNavigator` will try to match `Location` updates to a road: if a driver is too far away, the location cannot be matched.

> **Note**
> Note that previous events in the queue may still be delivered at least one time for the old route - as the events are delivered asynchronously. To prevent this, if desired, you can attach new listeners after setting the new route.

The Navigation example app available for Java and Kotlin shows how to detect the deviation.

The `RouteDeviation` event can be used to detect when a driver leaves the original route. Note that this can happen accidentally or intentionally, for example, when a driver decides while driving to take another route to the destination - ignoring the previous made choices for a route alternative and route options.

As shown above, you can detect the distance from the current location of the driver to the last known location on the route. Based on that distance, an application may decide whether it's time to calculate an entire new route or to guide the user back to the original route to keep the made choices for an route alternative and route options.

The HERE SDK does not recalculate routes automatically, it only notifies on the deviation distance - therefore any logic on how to get back to the route has to be implemented on app side.

> **Note**
> The `RouteDeviation` event will be fired for each new location update. To avoid unnecessary handling of the event, it may be advisable to wait for a few seconds to check if the driver is still deviating. If the event is no longer fired, it means that the driver is back on the route. Keep in mind that the route calculation happens asynchronously and that it is an app decision when and how to start a new route calculation. However, a new route can be set at any time during navigation to the `Navigator` or `VisualNavigator` instance and the upcoming events will be updated based on the newly set `Route` instance.

It is worth to mention that there can be also cases where a user is off-road. After a new route has been set, the user may still be off-road - therefore, the user has not been able to follow the route yet: in such a case you would still receive deviation events for the newly set route and `routeDeviation.lastLocationOnRoute` is null. If the current location of the user is not changing, it may be advisable to not start a new route calculation again.

The HERE SDK offers several APIs to react on a detected route deviation:

1. Recalculate the entire route with the `RoutingEngine` with new or updated `RouteOptions` to provide new route alternatives. If you use the current location of the user as new starting point, make sure to also specify a bearing direction for the first `Waypoint`.
2. Use the `returnToRoute()` method to calculate a new route to reach the originally chosen route alternative. It is available for the online `RoutingEngine` and the `OfflineRouteEngine`. Note that a route calculated with the `OfflineRouteEngine` does no longer include traffic information.
3. Refresh the old route with `routingEngine.refreshRoute()` using a new starting point that must lie on the original route and optionally update the route options. Requires a `RouteHandle` to identify the original route. This option does not provide the path from a deviated location back to the route, so it is not suitable for the deviation use case on its own.
4. On top, the HERE SDK offers the `DynamicRoutingEngine`, that allows to periodically request optimized routes based on the current traffic situation. It requires a route that was calculated online as it requires a `RouteHandle`. This is engine is meant to find better routes while the user is still following the route. Therefore, it may not be the best choice for the deviation use case, although it requires the current location as input.

The first and third option are covered in the Routing section. Note that the third option to refresh the original route does not provide the path from a deviated location back to the route. Therefore, it is not covered below. However, an application may choose to use it to subtract the travelled portion from the route and let users reach the new starting point on their own.

Based on parameters such as the distance and location of the deviated location an application needs to decide which option to offer to a driver.

However, the general recommendation is to use `returnToRoute()` when a deviation is detected as it will be the best option to route a user back to the original chosen route alternative - in case your app offers several route alternatives to be selected by a user.

> **Note**
> For each location update, the HERE SDK attempts to map-match the location to a street. Before processing a location update, its accuracy is evaluated upfront, and only locations deemed highly inaccurate are discarded. The location is then matched against the road network and any active route. Internally, various parameters are considered before a deviation event is triggered.

Return to a route after deviation

Calculate a route online or offline that returns to the original route with the `RoutingEngine` or the `OfflineRoutingEngine`. Use the `returnToRoute()` method when you want to keep the originally chosen route, but want to help the driver to navigate back to the route as quickly as possible.

> **Note**
> `returnToRoute()` is just one possible option to handle route deviations. See above for alternative options. For example, in some cases, it may be advisable to calculate an entire new route to the user's destination.

As of now, the `returnToRoute()` feature supports the same transport modes as the engine - you can use both, the `OfflineRoutingEngine` and the `RoutingEngine`. When executing the method with the `RoutingEngine`, only public transit routes are not supported - all other available transport modes for the `RoutingEngine` are supported.

> **Note**
> The `returnToRoute()` of the `OfflineRoutingEngine` method requires cached or already downloaded map data. In most cases, the path back to the original route may be already cached while the driver deviated from the route. However, if the deviation is too large, consider to calculate a new route instead.

When using the online `RoutingEngine`, it is required that the original `Route` contains a `RouteHandle` - or route calculation results in a `NO_ROUTE_HANDLE` error. For the `OfflineRoutingEngine` this is not necessary.

The route calculation requires the following parameters:

- The original `Route`, which is available from the `Navigator` / `VisualNavigator`.
- You will also need to set the part of the route that was already travelled. This information is provided by the `RouteDeviation` event.
- The new starting `Waypoint`, which may be the current map matched location of the driver.

The new starting point can be retrieved from the `RouteDeviation` event:

Java Kotlin

Explain this code

```
// Get current geographic coordinates.
val currentMapMatchedLocation = routeDeviation.currentLocation.mapMatchedLocation
val currentGeoCoordinates = currentMapMatchedLocation?.coordinates
    ?: routeDeviation.currentLocation.originalLocation.coordinates

// If too far away, consider to calculate a new route instead.
val newStartingPoint = Waypoint(currentGeoCoordinates); // See RouteDeviation.
```

With the online `RoutingEngine` it can happen that a completely new route is calculated - for example, when the user can reach the destination faster than with the previously chosen route alternative. The `OfflineRoutingEngine` preferably reuses the non-travelled portion of the route.

In general, the algorithm will try to find the fastest way back to the original route, but it will also respect the distance to the destination. The new route will try to preserve the shape of the original route if possible.

Stopovers that are not already travelled will not be skipped. For pass-through waypoints, there is no guarantee that the new route will take them into consideration at all.

Optionally, you can improve the route calculation by setting the heading direction of a driver:

Explain this code

```
if (currentMapMatchedLocation.bearingInDegrees != null) {
    newStartingPoint.headingInDegrees = currentMapMatchedLocation.bearingInDegrees;
}
```

Finally, we can calculate the new route:

Explain this code

```
routingEngine.returnToRoute(
        originalRoute,
        newStartingPoint,
        routeDeviation.lastTraveledSectionIndex,
        routeDeviation.traveledDistanceOnLastSectionInMeters, new CalculateRouteCallback() {
    @Override
    public void onRouteCalculated(@Nullable RoutingError routingError, @Nullable List<Route> list) {
        if (routingError == null) {
            Route newRoute = list.get(0);
            // ...
        } else {
            // Handle error.
        }
    }
});
```

Note
Since the `CalculateRouteCallback` is reused, a list of routes is provided. However, the list will only contain one route. The error handling follows the same logic as for the `RoutingEngine`.

As a general guideline for the online and offline usage, the `returnToRoute()` feature will try to reuse the already calculated portion of the `originalRoute` that lies ahead. Traffic data is only updated and taken into account when used with the online `RoutingEngine`.

The resulting new route will also use the same `OptimizationMode` as found in the `originalRoute`.

However, for best results, it is recommended to use the online `RoutingEngine` to get traffic-optimized routes.

Below you can find an example of a possible implementation using the above recommendations:

Explain this code

```
private void handleRerouting(RouteDeviation routeDeviation,
                             int distanceInMeters,
                             GeoCoordinates currentGeoCoordinates,
                             MapMatchedLocation currentMapMatchedLocation) {
    // Counts the number of received deviation events. When the user is following a route, no deviation
    // event will occur.
    // It is recommended to await at least 3 deviation events before deciding on an action.
    deviationCounter ++;

    if (isReturningToRoute) {
        // Rerouting is ongoing.
        Log.d(TAG, "Rerouting is ongoing ...");
        return;
    }

    // When user has deviated more than distanceThresholdInMeters. Now we try to return to the original route.
    int distanceThresholdInMeters = 50;
    if (distanceInMeters > distanceThresholdInMeters && deviationCounter >= 3) {
        isReturningToRoute = true;

        // Use current location as new starting point for the route.
        Waypoint newStartingPoint = new Waypoint(currentGeoCoordinates);

        // Improve the route calculation by setting the heading direction.
        if (currentMapMatchedLocation.bearingInDegrees != null) {
            newStartingPoint.headingInDegrees = currentMapMatchedLocation.bearingInDegrees;
        }

        // In general, the return.to-route algorithm will try to find the fastest way back to the original route,
        // but it will also respect the distance to the destination. The new route will try to preserve the shape
        // of the original route if possible and it will use the same route options.
        // When the user can now reach the destination faster than with the previously chosen route, a completely new
        // route is calculated.
        Log.d(TAG, "Rerouting: Calculating a new route.");
        routingEngine.returnToRoute(lastCalculatedRoute,
                                    newStartingPoint,
                                    routeDeviation.lastTraveledSectionIndex,
                                    routeDeviation.traveledDistanceOnLastSectionInMeters,
                                    (routingError, list) -> {
            // For simplicity, we use the same route handling.
            // The previous route will be still visible on the map for reference.
            handleRouteResults(routingError, list);
            // Instruct the navigator to follow the calculated route (which will be the new one if no error occurred).
            visualNavigator.setRoute(lastCalculatedRoute);
            // Reset flag and counter.
            isReturningToRoute = false;
            deviationCounter = 0;
            Log.d(TAG, "Rerouting: New route set.");
        });
    }
}
```

Note that this code uses a `deviationCounter` to not start rerouting too early. On top, it uses the flag `isReturningToRoute` to prevent that a new rerouting request is started before the previous one has completed.

The distance threshold is set to 50 m. If the user is still deviating from the route and the distance is greater than this threshold, rerouting will be considered.

With these checks you can call the method to handle a possible rerouting in the `RouteDeviation` callback - every time you receive a new deviation event. Note that when the user is following the route, then no deviation event is sent. On the other hand, when a new route was set after rerouting - based on the current location as new starting point - then no additional deviation event should occur, unless the user already deviated again. In this case, the rerouting process will be started again.