TomTom Go SDK Getting Started Guide

Table of Contents

What is GO SDK for Android?

TomTom's Android GO SDK is a convenient Software Development Kit that helps you make the most out of our Online Services in your mobile application, without the complexity of bare REST API calls. Optimized for Android applications, the GO SDK allows you to easily configure and deploy different location services within a single application.

The GO SDK is modular, allowing you to use only the features you need:

- Map Display SDK – displaying and interacting with beautiful maps in your application.
- Navigation SDK - enabling navigation-specific features and coordinating all aspects of the navigation process.
- Routing SDK – using an industry-leading routing engine to calculate routes with advanced parameters such as traffic avoidance, eco routes, reachable range, departure times, etc.

Getting Started

This GO SDK Example app is provided by TomTom, and is subject to the TomTom privacy policy at https://tomtom.com/privacy. Developers using TomTom SDKs and APIs in their apps are also responsible for adhering to all applicable privacy laws.

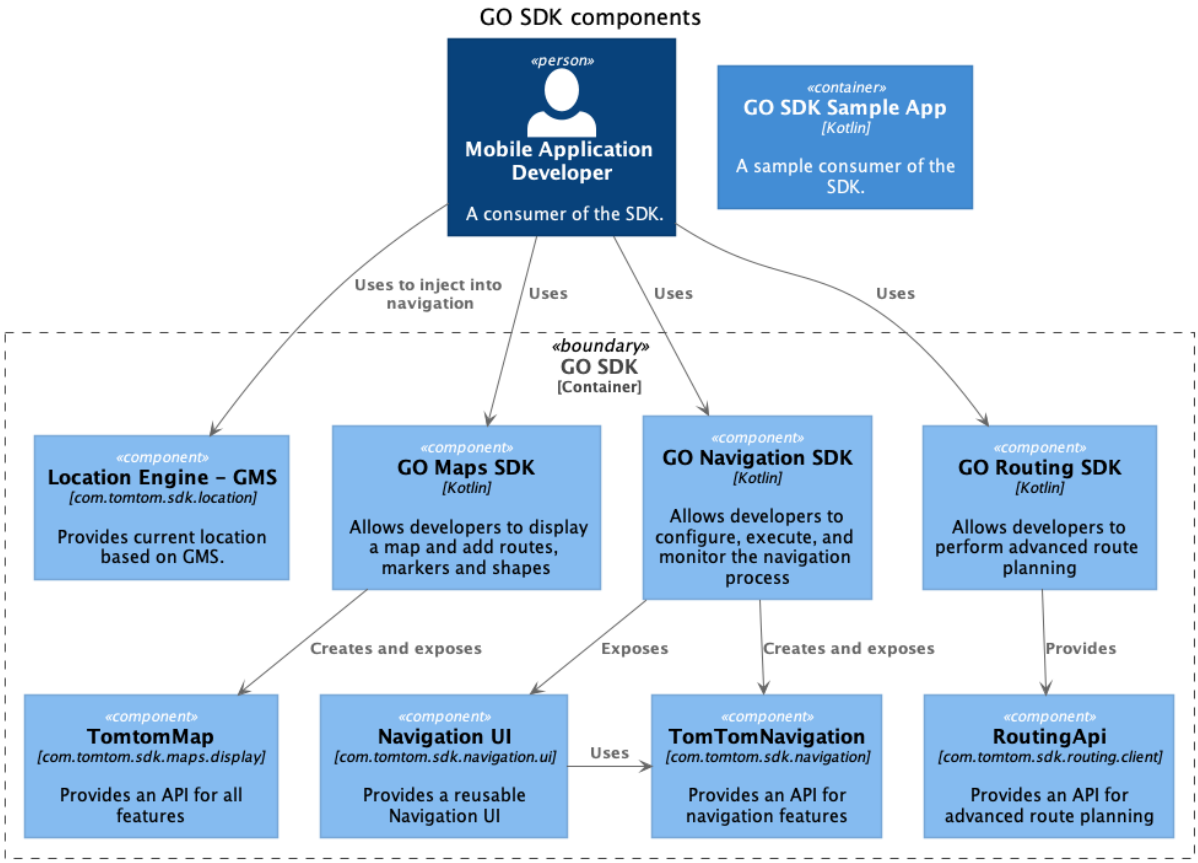
These GO SDK Examples are provided as-is. They are for internal use and evaluation purposes only. Any other use is strictly prohibited.

Common use cases to implement with the SDK:

- Add an interactive map to your application and overlay your own data, for example to build a store locator.
- Add advanced route planning to your application, including real time and predictive traffic, multiple alternatives, and avoidance criteria.

- Navigate along the planned route.

The diagram below shows main components delivered with the GO SDK:



Application setup

In the build.gradle file of your application module (e.g. app/build.gradle) make sure that:

- Java 8 support is enabled

```
android {
    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }
    packagingOptions {
        pickFirst "lib/**/libc++_shared.so"
    }
}
```

Map Display initialization

In order to use `MapFragment` add the following dependency to your project:

```
dependencies {
    implementation "com.tomtom.sdk:maps-display:$latest_version"
}
```

The Map Display module provides a ready-to-use `MapFragment` component.

It can be declared and instantiated either in the XML layout:

```
<androidx.fragment.app.FragmentContainerView
    xmlns:tomtom="http://schemas.android.com/apk/res-auto"
    android:id="@+id/map_fragment"
    android:name="com.tomtom.sdk.maps.display.ui.MapFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tomtom:mapKey="YOUR_API_KEY" />
```

(Note that usually the `tomtom` XML namespace, like any other XML namespace, is declared on the root-node level).

or via Kotlin/Java code:

```
fun createMapFragment(): MapFragment {
    val mapOptions = MapOptions(
        mapKey = "YOU_API_KEY_FOR_MAP",
        cameraOptions = cameraOptions,
        padding = Padding(5, 10, 5, 10),
        mapStyle = StyleDescriptor(
            uri = URI("http://path.to.your.style"),
            darkUri = URI("http://path.to.your.dark.style")
        )
    )

    return MapFragment.newInstance(mapOptions)
}
```

Once the `MapFragment` is instantiated, you can obtain the `TomTomMap` instance and use its API right away.

The map instance can be obtained asynchronously:

```
mapFragment.getMapAsync { tomtomMap: TomTomMap ->
    /* Your code goes here */
}
```

or via a synchronous call, as soon the `Fragment` is at least in the `STARTED` lifecycle phase:

```
```

Location engine initialization

In order to display where the user is on the map display or inform the navigation the current position to be able to generate guidance messages, location engines must be used.

The `AndroidLocationEngine` is used in real driving scenarios and requires a valid GPS fix.

In order to use `AndroidLocationEngine` add the following dependency to your project:

```
dependencies {
    implementation "com.tomtom.sdk:location-android:$latest_version"
}
```

For simulation or demo purposes, the SDK also provides a simulation engine: `SimulationLocationEngine`. It simulates location updates and provides coordinates which follow a predefined geometry, for example the currently planned route.

In order to use `SimulationLocationEngine` add the following dependency to your project:

```
dependencies {
    implementation "com.tomtom.sdk:location-simulation:$latest_version"
}
```

Which location engine to use with the map display can be selected in the following way:

```
val locationEngine = AndroidLocationEngine(context)
tomtomMap.setLocationEngine(locationEngine)
```

A current location marker will appear on the map display at the relevant position(s), a chevron image can also be selected for use during turn-by-turn navigation.

Use the engine that fits your application usage. Location engines can be switched at runtime. Note: The map display and navigation do not have to use the same location engine. In most cases you should use the AndroidLocationEngine for Navigation (or Simulation), yet you probably want the map display to show the chevron at the "map matched" location, not the real GPS location, in this case we also supply a `MapMatchedLocationEngine`, which can be used with the map display and takes a navigation object as input. Map matched positions generated by the navigation will then power the chevron for example.

Navigation initialization

The GO Navigation SDK consists of the two major components, `TomTomNavigation` and `NavigationFragment`.

In order to use `TomTomNavigation` and `NavigationFragment` add the following dependency to your project:

```
dependencies {
    implementation "com.tomtom.sdk:navigation:$latest_version"
    implementation "com.tomtom.sdk:navigation-ui:$latest_version"
}
```

`TomTomNavigation` allows developers to configure, execute, and monitor the navigation process:

```
val locationEngine = GmsLocationEngine(context)
val navigationConfigurationBuilder = NavigationConfiguration.Builder(
    context = context,
    navigationApiKey = "NAVIGATION_API_KEY",
    locationEngine = locationEngine
)
val navigationConfiguration = navigationConfigurationBuilder.build()
val navigation = TomTomNavigation.create(navigationConfiguration)
```

Meanwhile, the `NavigationFragment` provides reusable navigation UI components:

```
val navigationUiOptions = NavigationUiOptions.Builder()
    .units(Units.METRIC)
    .build()

val navFragment: NavigationFragment = NavigationFragment.newInstance
(navigationUiOptions)
supportFragmentManager.beginTransaction()
    .add(R.id.navigation_fragment_container, navFragment)
    .commitNow()

navFragment.setTomTomNavigation(navigation)
```

Navigation fragment

`NavigationFragment` is an Android fragment that wraps the `TomTomNavigation`. It provides the UI that shows information about the oncoming manoeuvres, route parameters (like ETA and remaining distance), current speed and the speed limit. It can be instantiated either via the XML view inflation or programmatically as it is shown in `Navigation initialization` section.

After setting an instance of `TomTomNavigation` (that should be called before calling any other navigation method) you can:

- `startNavigation(routes: List<Route>)` - this method wraps the call to `TomTomNavigation.start`. When called, the UI starts showing `Route`'s guidance, current speed and speed limit. `NavigationListener.onStarted` is called when navigation has been started.
- `updateRoute(route: Route)` - this method wraps the call to `TomTomNavigation.update`. If `startNavigation(routes: List<Route>)` has been called already, the UI will change accordingly.
- `stopNavigation()` - this method wraps the call to `TomTomNavigation.stop`. When called, the UI stops showing `Route`'s guidance, current speed and speed limit.

Note: Disposal of `TomTomNavigation` is not handled on the `NavigationFragment` side, so `TomTomNavigation.dispose` has to be called by you when `TomTomNavigation` is no longer needed.

`NavigationFragment` also exposes `NavigationListener` with the following methods:

- `onStarted()` - called when navigation has been started.
- `onFailed(exception: NavigationException)` - called when some exception happened while navigation was running. The `NavigationException` contains a description what exactly went wrong.
- `onCancelled()` - called when cancel button has been clicked.
- `changeTextToSpeechEngine(ttsEngine: TextToSpeechEngine)` - use this method to change the implementation used by underlying `TextToSpeech`

```
val navListener = object : NavigationFragment.NavigationListener {
    override fun onStarted() {
        /* Your code goes here */
    }

    override fun onFailed(exception: NavigationException) {
        /* Your code goes here */
    }

    override fun onCancelled() {
        /* Your code goes here */
    }
}
```

You can register on these events by calling:

```
navFragment.addNavigationListener(navListener)
```

When you do not need it any more, call:

```
navFragment.removeNavigationListener(navListener)
```

Navigating without the route

Navigating without the route is a mode when we start navigation without destination. In that case, only current device location is matched to the map and detailed information about a current location on the road are provided. This mode can be useful for drivers who know the area and do not need turn-by-turn navigation. For example, they just want to know their speed and current limits.

The GO Navigation SDK allows to start navigation without the route by calling `start` method without parameter:

```
navigation.start()
navigation.addOnNavigationStartedListener(object : OnNavigationStartedListener {
    override fun onNavigationStarted(routes: List<Route>) {
        /* Your code goes here */
    }
})
navigation.addOnNavigationErrorListener(object : OnErrorListener {
    override fun onError(exception: NavigationException) {
        /* Your code goes here */
    }
})
navigation.addOnLocationMatchedListener(object : OnLocationMatchedListener {
    override fun onLocationMatched(location: GeoLocation) {
        /* Your code goes here */
    }
})
navigation.addOnLocationContextUpdateListener(object :
OnLocationContextUpdateListener {
    override fun onLocationContextUpdated(locationContext: LocationContext) {
        /* Your code goes here */
    }
})
```

When navigation is no longer needed you should dispose it, see Stop navigation section.

Navigating without the route lifecycle

If `NavigationOptions` does not contain any `Route` then navigation will proceed with following steps:

- Providing `MatchedLocation` on `OnLocationMatchedListener`
- Providing `LocationContext` on `OnLocationContextUpdateListener`

Note: More information about listeners can be found under Navigation callbacks section.

Navigation along the route

Navigation along the route is a turn-by-turn navigation mode. In that mode we provided a custom route that we want the driver to follow. A custom route can be retrieved using the GO Routing SDK, see `Request a route` section. After starting navigation with list of routes the navigation will go through with full lifecycle. You can subscribe to steps that interest you, such as route progress update, new guidance or destination reach event.

```
val navigationOptions = NavigationOptions(routes)
navigation.start(navigationOptions)
navigation.addOnNavigationStartedListener(object : OnNavigationStartedListener {
    override fun onNavigationStarted(routes: List<Route>) {
        /* Your code goes here */
    }
})
navigation.addOnNavigationErrorListener(object : OnErrorListener {
    override fun onError(exception: NavigationException) {
```

```
        /* Your code goes here */
    }
})
navigation.addOnLocationMatchedListener(object : OnLocationMatchedListener {
    override fun onLocationMatched(location: GeoLocation) {
        /* Your code goes here */
    }
})
navigation.addOnLocationContextUpdateListener(object :
OnLocationContextUpdateListener {
    override fun onLocationContextUpdated(locationContext: LocationContext) {
        /* Your code goes here */
    }
})
navigation.addOnProgressUpdateListener(object : OnProgressUpdateListener {
    override fun onProgressUpdated(progress: RouteProgress) {
        /* Your code goes here */
    }
})
navigation.addOnRouteDeviationListener(object : OnRouteDeviationListener {
    override fun onRouteDeviated(location: GeoLocation, route: Route) {
        /* Your code goes here */
    }
})
navigation.addOnRouteRefreshListener(object : OnRouteRefreshListener {
    override fun onRouteRefreshRequested(location: GeoLocation, currentRoute:
Route) {
        /* Your code goes here */
    }
})
navigation.addOnGuidanceUpdateListener(object : OnGuidanceUpdateListener {
    override fun onInstructionsChanged(instructions: List<GuidanceInstruction>)
{
        /* Your code goes here */
    }

    override fun onAnnouncementGenerated(announcement: GuidanceAnnouncement) {
        /* Your code goes here */
    }

    override fun onDistanceToCurrentInstructionChanged(
        distance: Double,
        instructions: List<GuidanceInstruction>
    ) {
        /* Your code goes here */
    }
})
navigation.addOnDestinationReachedListener(object : OnDestinationReachedListener
{
    override fun onDestinationReached() {
        /* Your code goes here */
    }
})
```

When navigation is no longer needed you should dispose it, see Stop navigation section.

Navigation along the route lifecycle

If `NavigationOptions` contains at least one `Route` then navigation will go through all steps:

- Providing `MatchedLocation` on `OnLocationMatchedListener`
- Providing `LocationContext` on `OnLocationContextUpdateListener`
- Calculation of `RouteProgress` on `OnProgressUpdateListener`
- Route deviation check with possible call on `OnRouteDeviationListener`
- Check for route refresh with possible call on `OnRouteRefreshListener`
- Generation of `Guidance` on `OnGuidanceUpdateListener`
- Detection of arrival with possible call on `OnDestinationReachedListener`

Note: More information about listeners can be found under Navigation callbacks section.

Update route

In some cases you may need to `update` a current navigation process with new `Route`. This could be a situation when you start a navigation without the route and then decided to switch to navigation along the route, or you got `onRouteDeviated` event and want to present user a new available `Route`. There is also a situation when it is recommended, when you got `onRouteRefreshRequested` event, see Navigation callbacks.

```
val navigationOptions = NavigationOptions(routes)
navigation.start(navigationOptions)
navigation.addOnRouteDeviationListener(object : OnRouteDeviationListener {
    override fun onRouteDeviated(location: GeoLocation, route: Route) {
        val newRoute = requestNewRoute(location)
        navigation.update(newRoute)
    }
})
```

Stop navigation

To stop current navigation process just call `stop` method on the `TomTomNavigation` instance:

```
navigation.stop()
```

When navigation is no longer needed it should be disposed:

```
navigation.dispose()
```

Navigation callbacks

`TomTomNavigation` exposes its lifecycle events that user can subscribe to:

- `OnNavigationStartedListener` informs about the successful navigation start.

```
val listener = object : OnNavigationStartedListener {
    override fun onNavigationStarted(routes: List<Route>) {
        /* Your code goes here */
    }
}
navigation.addOnNavigationStartedListener(listener)
```

- `OnErrorListener` informs about errors that occurred during navigating.

```
navigation.addOnNavigationErrorListener(object : OnErrorListener {
    override fun onError(exception: NavigationException) {
        /* Your code goes here */
    }
})
```

- `OnLocationMatchedListener` informs about new location that was matched to map.

```
navigation.addOnLocationMatchedListener(object : OnLocationMatchedListener {
    override fun onLocationMatched(location: GeoLocation) {
        /* Your code goes here */
    }
})
```

- `OnLocationContextUpdateListener` informs about detailed information in current location on road.

```
navigation.addOnLocationContextUpdateListener(object :
OnLocationContextUpdateListener {
    override fun onLocationContextUpdated(locationContext: LocationContext) {
        /* Your code goes here */
    }
})
```

- `OnProgressUpdateListener` informs about the new progress along the route.

```
navigation.addOnProgressUpdateListener(object : OnProgressUpdateListener {
    override fun onProgressUpdated(progress: RouteProgress) {
        /* Your code goes here */
    }
})
```

- `OnRouteDeviationListener` informs about getting off the route.

```
navigation.addOnRouteDeviationListener(object : OnRouteDeviationListener {
    override fun onRouteDeviated(location: GeoLocation, route: Route) {
        /* Your code goes here */
    }
})
```

- `OnRouteRefreshListener` informs about need of refreshing the current route. It might happens when navigation needs new estimated time of arrival. Whenever it happens then it is recommended to provide new `Route` through `TomTomNavigation.update` method.

```
navigation.addOnRouteRefreshListener(object : OnRouteRefreshListener {
    override fun onRouteRefreshRequested(location: GeoLocation, currentRoute:
Route) {
        /* Your code goes here */
    }
})
```

- `OnGuidanceUpdateListener` informs about the new guidance.

```
navigation.addOnGuidanceUpdateListener(object : OnGuidanceUpdateListener {
    override fun onInstructionsChanged(instructions: List<GuidanceInstruction>)
{
        /* Your code goes here */
    }

    override fun onAnnouncementGenerated(announcement: GuidanceAnnouncement) {
        /* Your code goes here */
    }

    override fun onDistanceToCurrentInstructionChanged(
        distance: Double,
        instructions: List<GuidanceInstruction>
    ) {
        /* Your code goes here */
    }
})
```

- `OnDestinationReachedListener` informs about reaching destination. By default, it is triggered when the distance to the destination is 50 meters or less or the arrival time is 30 seconds or less.

```
navigation.addOnDestinationReachedListener(object : OnDestinationReachedListener
{
    override fun onDestinationReached() {
        /* Your code goes here */
    }
})
```

- `OnRouteUpdatedListener` informs about the successful update of the route.

```
navigation.addOnRouteUpdatedListener(object : OnRouteUpdatedListener {
    override fun onRouteUpdated(route: Route) {
        /* Your code goes here */
    }
})
```

Note: It is recommended to remove all added listeners if they are no longer needed. It can be done via calling proper method for all kinds of added callbacks e.g.,

```
navigation.removeOnNavigationStartedListener(listener)
```

Routing initialization

In order to use `RoutingApi` add the following dependency to your project:

```
dependencies {
    implementation "com.tomtom.sdk:routing-client:$latest_version"
}
```

The GO Routing SDK module provides the `RoutingApi` that is the default TomTom implementation to perform route planning action. It can be initialized in the following way:

```
val routingApi = RoutingApi.create(context, ROUTING_API_KEY)
```

Request a route

In order to obtain the route(s) between two points it is necessary to use the `RoutingOptions` instance and initialize it with the origin and the destination coordinates. The `RoutingOptions` offers a variety of other options that help in finetuning the route retrieval.

```
val amsterdamCoordinates = GeoCoordinate(52.3772547, 4.9097686)
val berlinCoordinates = GeoCoordinate(52.4954702, 13.4619186)
val routingOptions = RoutingOptions(amsterdamCoordinates, berlinCoordinates)
val routes = routingApi.planRoute(routingOptions).value().routes
```

The route obtained from this module can be loaded to the Navigation module for further processing.

Routing for trucks

In order to request a route for trucks you need to extend `RoutingOptions` by `TravelMode` and `CombustionVehicleDescriptor` parameters. The first parameter should be set to `TravelMode.TRUCK`. The second one is more complex and contains all the detailed vehicle data (all of them are optional):

- `VehicleDimensions` specifies the dimensions of the vehicle:

  - `vehicleWeightInKg` - the weight of the vehicle in kilograms.
  - `vehicleAxleWeightInKg` - the weight per axle of the vehicle in kilograms.
  - `vehicleLengthInMeters` - the length of the vehicle in meters.
  - `vehicleWidthInMeters` - the width of the vehicle in meters.
  - `vehicleHeightInMeters` - the height of the vehicle in meters.
- `VehicleEfficiency` specifies the efficiency of the vehicle:
  - `uphillEfficiency` - the efficiency of converting chemical energy stored in fuel to potential energy when the vehicle gains elevation.
  - `downhillEfficiency` - the efficiency of converting potential energy to saved (not consumed) fuel when the vehicle loses elevation.
  - `accelerationEfficiency` - the efficiency of converting chemical energy stored in fuel to kinetic energy when the vehicle accelerates.
  - `decelerationEfficiency` - the efficiency of converting kinetic energy to saved (not consumed) fuel when the vehicle decelerates.
- `VehicleRestrictions` specifies the restriction that this vehicle should be subjected to:
  - `vehicleMaxSpeedInKph` - maximum speed of the vehicle in km/hour.
  - `isVehicleCommercial` - determines whether the vehicle is used for commercial purposes and thus may not be allowed to drive on some roads.
  - `vehicleLoadType` - represents types of cargo that may be classified as hazardous materials and restricted from some roads. Available `vehicleLoadType` values are US Hazmat classes 1 through 9, plus generic classifications for use in other countries:
    - `UNDEFINED` - the vehicle load type is not set.
    - `US_HAZMAT_CLASS_1` - the cargo contains explosives.
    - `US_HAZMAT_CLASS_2` - the cargo contains hazardous gases.
    - `US_HAZMAT_CLASS_3` - the cargo contains flammable and combustible liquids.
    - `US_HAZMAT_CLASS_4` - the cargo contains flammable solids.
    - `US_HAZMAT_CLASS_5` - the cargo contains oxidizing substances and organic peroxides.
    - `US_HAZMAT_CLASS_6` - the cargo contains toxic substances and infectious substances.
    - `US_HAZMAT_CLASS_7` - the cargo contains radioactive materials.
    - `US_HAZMAT_CLASS_8` - the cargo contains corrosives.
    - `US_HAZMAT_CLASS_9` - the cargo contains miscellaneous hazardous materials.
    - `OTHER_HAZMAT_EXPLOSIVE` - the cargo contains explosive materials other than in class 1.
    - `OTHER_HAZMAT_GENERAL` - the cargo contains other general hazardous materials.
    - `OTHER_HAZMAT_HARMFUL_TO_WATER` - the cargo contains other hazardous substances that are harmful to water.
  - `vehicleAdrTunnelRestrictionCode` - if specified, the vehicle is subject to ADR tunnel restrictions:
    - `UNDEFINED` - the vehicle ADR tunnel restrictions are not set.
    - `B` - vehicles with code B are restricted from roads with ADR tunnel categories B, C, D, and E.
    - `C` - vehicles with code C are restricted from roads with ADR tunnel categories C, D, and E.
    - `D` - vehicles with code D are restricted from roads with ADR tunnel categories D, and E.
    - `E` - vehicles with code E are restricted from roads with ADR tunnel categories E.
- `CombustionVehicleConsumption` describes parameters used to determine the vehicle power and consumption:
  - `currentFuelInLiters` - current fuel in liters.
  - `auxiliaryPowerInLitersPerHour` - the amount of fuel consumed for sustaining

auxiliary systems of the vehicle, in liters per hour. It can be used to specify consumption due to devices and systems such as AC systems, radio, heating, etc.

- `fuelEnergyDensityInMJoulesPerLiter` - the amount of chemical energy stored in one liter of fuel in megajoules (MJ). It is used in conjunction with the *Efficiency parameters for conversions between saved or consumed energy and fuel. For example, energy density is 34.2 MJ/l for gasoline, and 35.8 MJ/l for Diesel fuel.
- `speedConsumptionInLitersPerHundredKm` - map used to determine fuel consumption at different speeds.

Here is an example of `RoutingOptions` that can be used to request the route for trucks:

```
val amsterdamCoordinates = GeoCoordinate(52.3772547, 4.9097686)
val berlinCoordinates = GeoCoordinate(52.4954702, 13.4619186)

val truckDimensions = VehicleDimensions.Builder()
    .vehicleWeightInKg(25000)
    .vehicleAxleWeightInKg(5000)
    .vehicleLengthInMeters(13.6)
    .vehicleWidthInMeters(2.5)
    .vehicleHeightInMeters(4.0)
    .build()

val truckRestrictions = VehicleRestrictions.Builder()
    .vehicleMaxSpeedInKph(90)
    .isVehicleCommercial(true)
    .vehicleLoadType(VehicleLoadType.US_HAZMAT_CLASS_1)
    .vehicleAdrTunnelRestrictionCode(VehicleAdrTunnelRestrictionCode.D)
    .build()

val truckConsumption = CombustionVehicleConsumption(
    currentFuelInLiters = 350.0,
    auxiliaryPowerInLitersPerHour = 5.0,
    fuelEnergyDensityInMJoulesPerLiter = 35.8,
    speedConsumptionInLitersPerHundredKm = mapOf(90.0 to 20.0)
)

val truckVehicleDescriptor = CombustionVehicleDescriptor.Builder()
    .vehicleDimensions(truckDimensions)
    .vehicleRestrictions(truckRestrictions)
    .vehicleConsumption(truckConsumption)
    .build()

val routingOptions = RoutingOptions(
    origin = amsterdamCoordinates,
    destination = berlinCoordinates,
    travelMode = TravelMode.TRUCK,
    combustionVehicleDescriptor = truckVehicleDescriptor
)
```

Note: More information about `CombustionVehicleDescriptor` parameters can be found under Routing API documentation page.

Routing with extended guidance

It is possible to obtain the desired route with some additional information considering guidance. In order to that you can specify the following parameters in `RoutingOptions`.

```
val amsterdamCoordinates = GeoCoordinate(52.3772547, 4.9097686)
val berlinCoordinates = GeoCoordinate(52.4954702, 13.4619186)
val routingOptions = RoutingOptions(
    origin = amsterdamCoordinates,
    destination = berlinCoordinates,
    instructionAnnouncementPoints = AnnouncementPoints.ALL,
    instructionPhonetics = InstructionPhoneticsType.IPA,
    instructionRoadShieldReferences = RoadShieldReferences.ALL
)
```

The additional parameters:

- instructionAnnouncementPoints - If specified, the instruction will include up to three additional fine-grained announcement points, each with their own location, maneuver type, and distance to the instruction point.
- instructionPhonetics - Specifies whether to include the phonetic transcription of street names, signpost text, and road numbers in the instructions of the response.
- instructionRoadShieldReferences - Specifies whether to include road shield references into the external road shields atlas.

To use this method, you need an API Key that supports routing with extended guidance.

TTS

Text-To-Speech (TTS) functionality is provided via the `TextToSpeech` class. To use `TextToSpeech`, you need to add the following dependency to your project first:

```
implementation "com.tomtom.sdk:tts:$latest_version"
```

Initialization

An instance can be created with the default engine, which is based on Android's TextToSpeech.

```
val tts = TextToSpeech(context)
```

A custom engine can be passed as well.

```
val customEngineTts = TextToSpeech(ttsEngine)
```

From now on, `TextToSpeech` will use the passed engine to synthesize voice.

To listen for whether the engine is ready, register the `OnEngineReadyListener`. If more than one

component needs to wait for the engine to be ready, you can add multiple instances of `OnEngineReadyListener`.

```
val listener = object : OnEngineReadyListener {
    override fun onReady() {
        // Your code here
    }

    override fun onError(exception: TextToSpeechEngineException) {
        // Your code here
    }
}
tts.addOnEngineReadyListener(listener)
```

Remove the `OnEngineReadyListener` when it is no longer needed. `OnEngineReadyListeners` will also be automatically removed when `dispose()` is called.

```
tts.removeOnEngineReadyListener(listener)
```

Message synthesis

This section describes the behavior of the default Android-based `TextToSpeechEngine`

To synthesize an audio message, use the `playAudioMessage` method. The `AudioMessagePlaybackListener` can be used to track the status of voice synthesis.

```
val audioMessage = AudioMessage("In 300 meters turn left", MessageType.PLAIN)
val messagePlaybackListener = object : MessagePlaybackListener {
    override fun onStart() {
        // Your code here
    }

    override fun onDone() {
        // Your code here
    }

    override fun onError(exception: TextToSpeechEngineException) {
        // Your code here
    }

    override fun onStop() {
        // Your code here
    }
}
tts.playAudioMessage(audioMessage, MessageConfig(0, 10_000),
messagePlaybackListener)
```

Audio messages formatted in SSML are also supported by the `playAudioMessage` method.

```
val ssmlMessage = AudioMessage(
    "<speak>Turn left onto <phoneme alphabet='ipa' ph='e¬¬.¬f¬¬¬'>A4</phoneme>
towards " +
        "<phoneme alphabet='ipa'
ph=''sxep.fart.my.'2ze.^m'>Scheepvaartmuseum</phoneme></speak>",
    MessageType.SSML
)
```

You can also pass tagged messages with phonetics to be substituted with the `playTaggedMessage` method. An example `TaggedMessage` looks like this:

```
val roadNumberPhonetics = PhoneticTranscription(
    listOf("e¬¬.¬f¬¬¬"),
    listOf("nl-NL"),
    "roadNumber",
    "ipa"
)
val signpostPhonetics = PhoneticTranscription(
    listOf("'sxep.fart.my.'2ze.^m"),
    listOf("nl-NL"),
    "signpostText",
    "ipa"
)

val taggedMessage = TaggedMessage(
    "Turn left onto <roadNumber>A4</roadNumber> towards
<signpostText>Scheepvaartmuseum</signpostText>",
    listOf(roadNumberPhonetics, signpostPhonetics)
)
```

It is synthesized similarly to `AudioMessage`, but with a separate method:

```
val taggedMessagePlaybackListener = object : MessagePlaybackListener {
    override fun onStart() {
        // Your code here
    }

    override fun onDone() {
        // Your code here
    }

    override fun onError(exception: TextToSpeechEngineException) {
        // Your code here
    }

    override fun onStop() {
        // Your code here
    }
}
tts.playTaggedMessage(taggedMessage, MessageConfig(0, 10_000),
taggedMessagePlaybackListener)
```

Underneath, the engine parses the provided message to SSML format (after parsing it looks like this) and synthesizes it.

It is also possible to change the language of the underlying engine.

```
tts.changeLanguage(Locale.forLanguageTag("pl-PL"))
```

Queuing

Message queuing depends on message priority. If the message that is currently being synthesized has an equal or higher priority to the new message, the new message will be added to the queue (taking the priorities of queued messages into account). If the message that is currently being synthesized has a lower priority than the new one, it will be interrupted and the new message will be processed right away.

Cancelling

When either `playAudioMessage` or `playTaggedMessage` method is called, an instance of `Cancellable` is returned. It can be used to cancel the operation. It will be then removed from the queue or, if processing has already started, its processing will be stopped.

```
val cancellable = tts.playAudioMessage(audioMessage, MessageConfig(0, 10_000),
messagePlaybackListener)
cancellable.cancel()
```

Disposal

Dispose of the resources used by `TextToSpeechEngine` when it is no longer needed.

```
tts.dispose()
```

After `dispose()` has been called, audio messages cannot be synthesized. An `EngineNotReadyException` exception will be returned instead. Create a new instance to synthesize further audio messages.