

EX1025



HERE SDK for iOS

Developer's Guide

Version 4.0.0.0

Legal Notices

© 2019 HERE Global B.V. and its Affiliate(s). All rights reserved.

This material, including documentation and any related computer programs, is protected by copyright controlled by HERE. All rights are reserved. Copying, including reproducing, storing, adapting or translating, any or all of this material requires the prior written consent of HERE. This material also contains confidential information, which may not be disclosed to others without the prior written consent of HERE.

Trademark Acknowledgements

HERE is trademark or registered trademark of HERE Global B.V. Other product and company names mentioned herein may be trademarks or trade names of their respective owners.

Disclaimer

This content is provided "as-is" and without warranties of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability, fitness for a particular purpose, satisfactory quality and non-infringement. HERE does not warrant that the content is error free and HERE does not warrant or make any representations regarding the quality, correctness, accuracy, or reliability of the content. You should therefore verify any information contained in the content before acting on it.

To the furthest extent permitted by law, under no circumstances, including without limitation the negligence of HERE, shall HERE be liable for any damages, including, without limitation, direct, special, indirect, punitive, consequential, exemplary and/ or incidental damages that result from the use or application of this content, even if HERE or an authorized representative has been advised of the possibility of such damages.

Document Information

Product

Name: HERE SDK for iOS

Version: Version 4.0.0.0

Document

Name: HERE SDK for iOS - Developer's Guide

ID: 0b187bbc-fe93-4a44-a162-6c91abb25721

Status: FINAL

Date: 2019-07-16T21:15:08.532Z

Table of Contents

Overview

Get Started

[Minimum Requirements](#)

[Supported Devices](#)

[Deployment Requirements](#)

[Creating a new iOS Project](#)

[Say Hello Map!](#)

[Integrating the HERE SDK](#)

[Setting your HERE credentials](#)

[Adding the Map View](#)

[Loading a Map Scene](#)

[Prepare the framework for distribution](#)

[Troubleshooting](#)

Use Cases

General Concepts

[How to use this guide?](#)

[Engines](#)

[Custom Map Styles](#)

[Camera](#)

[Gestures](#)

Map Items

[Add Map Polylines](#)

[Add Map Polygons](#)

[Add Map Circles](#)

[Add Map Markers](#)

[Add Map Overlays](#)

Search

[Places Search](#)

[Reverse Geocoding an Address from Coordinates](#)

[Geocoding an Address to a Location](#)

Traffic

Directions

Coverage Information

API Reference

Examples

Overview

Document Version: 2019-07-16T21:15:08.532Z

HERE SDK for iOS: Version 4.0.0.0

What is the HERE SDK for iOS?

The new HERE SDK for iOS enables you to build powerful maps applications. Bundled along with it are many of HERE's assets, available on iOS for customers to integrate with their own apps. The HERE SDK for iOS consumes data from HERE's Open Location Platform (OLP) incorporating microservices and highly modularized components. The HERE SDK for iOS belongs to the HERE SDK product family which also includes the *HERE SDK for Android*.

Backed by more than 30 years of HERE's expertise in mapping technology, the new lightweight HERE SDK for iOS gives you instant access to the freshest map data of the highest quality, consistency and accuracy. Build immersive native applications and benefit from the world's most accurate and interactive map data that is captured via a comprehensive and precise industrial process with HERE's True vehicles.

Explore maps that are fast and smooth to interact with, pan/zoom across map views of varying resolutions, and enable the display of various elements such as routes and POIs on highly customizable map views.

We are happy to hear your feedback. Thank you for using the HERE SDK for iOS!

Feature List

At the time of writing, the HERE SDK for iOS, version 4.0.0.0 is the current release, including:

Mapping

- Latest map data from HERE's Open Location Platform for more than 190 countries with weekly map updates.
- Resource-friendly vector tile map rendering.
- 2D / 2.5D rendering including building footprints, extruded buildings, carto POIs, and more.
- Highly customizable map styles featuring a powerful WYSIWYG customization tool, including predefined styles such as normal street map and satellite map.
- Support for all common map interactions and gestures (including pan, flick, and pinch zoom).
- Pickable overlay objects on the map such as polylines, polygons, circles, images, and routes.
- Attachable native iOS views as map overlays.
- Highly customizable map marker support (including anchor points and multiple images).
- Metadata support for most map items (for example, to identify search results on the map).

Search

- Search through a broad set of geographical content across the globe (including streets, address points, and places).
- Access localized details for a Point of Interest (POI).
- Perform geocoding and reverse geocoding lookups.
- All address details are already localized and can be accessed in various languages.

Traffic

- Get real-time traffic information for nearby and remote traffic incidents.

- Embedded support for instant visualization of traffic flows on the map.
- Embedded support for instant visualization of traffic incidents on the map.
- All traffic information is already localized and can be accessed in various languages.

Directions

- Online car route directions.
- Online pedestrian route directions.
- Online truck route directions with highly customizable truck options.
- Online bicycle route directions (BETA stage).
- Specify preferred route type (fastest or shortest) and route options (such as speed profiles, route restrictions, vignette options, and more).
- Calculate traffic-aware routes indicating potential traffic delay times, as well as traffic-optimized routes to avoid traffic.
- Customizable route shapes utilizing different types of waypoints.
- Localized maneuver instructions are available for each route.
- Alternate routes.

The HERE SDK for iOS can seamlessly be integrated into the car's built-in display using Apple's [CarPlay](#).

Where to Start?

If you are interested in building your first app showing a HERE map, please have a look at our [Get Started](#) guide to easily guide you through the first simple steps.

Additional resources:

- In the [example section](#), you can find the links to our example applications that accompany this user guide.
- The API Reference can be found on [developer.here.com](#) or bundled on our [S3 bucket](#).

On our [S3 bucket](#) you can find all current *release artifacts* including the HERE SDK framework to include in your application(s). We also host nightly builds (named *generation.major.minor.patch.nightly_date*) to reflect the ongoing development process - these builds might be unstable and should be used at your own risk. We recommend to use the latest released artifacts (named *generation.major.minor.patch.release_date*), as only these builds are guaranteed to be verified by a series of manual and automated tests.

Legal Requirements

The use of the HERE SDK for iOS is governed by the terms and conditions which you have accepted separately, either when signing up via the [developer.here.com](#) portal or otherwise executing an agreement with HERE. For your use of the documentation presented on this website, [these terms](#) apply.

In addition to the above terms, the following shall apply: Components of the HERE SDK for iOS collect certain information from your application. Such information includes access credentials (`AccessKeyId` , `AccessKeySecret` – see also [Authenticating Applications](#)) and the types of features utilized by your application when used by end users. The information does not identify an individual end user. However, your application's privacy policy must disclose to the end users that you have licensed products and services from HERE, that such information is collected from your application as it is being used by end users, and that HERE collects and processes such information from the application.

Get Started

Thank you for using the HERE SDK for iOS. Let us introduce you to the HERE SDK and show you how easy it is to get started. As a very first example, we will develop a "Hello Map" iOS application that shows - yes! - a map.

Minimum Requirements

- The HERE SDK for iOS requires iOS 11.0 or higher.
- As absolute minimum, you must use Xcode 10.2 and Swift 5 for development.
- The accompanying example apps have been developed using the minimum requirements.

Supported Devices

The HERE SDK supports only iOS ARM64 architectures, which means that all iOS devices starting from the iPhone 5s (which was released in 2013) are supported. The x86_64 architecture is supported for simulators.

Deployment Requirements

The HERE SDK framework is a **fat binary**, built for device (arm64) *and* simulator (x86_64). *Therefore it contains both architectures. This allows easy deployment on a simulator _and_ on a real device during the development phase of your app.* However, when you want to archive your project for distribution, please make sure to remove the included x86_64 architecture, as described [here](#).

Creating a new iOS Project

Let's see how easy it is to create a new iOS app showing a HERE map. If you want to integrate the SDK into an existing application, you can skip this step. No specific SDK code is involved here. We recommend using Xcode as the IDE. If you are new to iOS development, please follow the guides on developer.apple.com to help you get started with the first steps.

Note: The example code for "HelloMap" is available from [here](#).

Start Xcode (for this guide, we have used version 10.2.1) and *Create a new Xcode project*:

- For the template, it is sufficient to choose *iOS => Application => Single View App*.
- Click *next*.
- Provide a project name, e.g. "HelloMap" and select *Swift* as the language.
- Click *next* and choose an appropriate directory in which to save the project.
- Select a simulator or a real device (which is recommended).

For the example below, we have kept the default orientation settings. When running the existing example apps, please make sure the *Deployment target* is set to a version you have installed - we have used iOS 11.

Now, *build and run the current scheme* and confirm that your plain project executes as expected. If all goes well, you should see a blank view without any content.

Say Hello Map!

Once we have a working iOS app, it's time to include the HERE SDK and to show a map on your emulator or device - in just four simple steps:

1. Integrate the HERE SDK.
2. Set required credentials in your `plist` file.
3. Add a `MapView` by code or by adding it to a storyboard.
4. Add the code to load your first map scene.

Note: While all of the functionalities of the HERE SDK are accessible from the simulator, usage of a real device is strongly recommended. The overall performance will be better, and some features like gestures are just easier to use on a multi-touch enabled hardware.

Integrating the HERE SDK

On developer.here.com or on our [S3 bucket](#) you can find the latest *release artifacts* including the HERE SDK framework to include in your application (named `xx.yy.zz.release_date`).

Copy the `heresdk.framework` file to your app's root folder.

In Xcode, open the *General* settings of the *App* target and add the `heresdk.framework` to the *Embedded Binaries* section (Click "+", then "Add other..." -> "Create folder references").

Note that this guide is based on HERE SDK version 4.0.0.0. If your framework version is different from the version used for this guide, you may need to adapt the example's source code.

Setting your HERE credentials

When using the HERE SDK, your application must be authenticated with a set of credentials you add to the `Info.plist` file of your project. For this, you need to acquire a set of credentials by registering yourself on developer.here.com - or ask your HERE representative.

Once you have your credentials at hand, add a new *Dictionary* entry called "HERECredentials" to your *Information Property List* file (`Info.plist`). Inside "HERECredentials", add the following key/value pairs for `AccessKeyId` and `AccessKeySecret`. Alternatively, you can also open the file "as source" and add the following XML tags inside the existing `<dict>` element:

```
<key>HERECredentials</key>
<dict>
  <key>AccessKeyId</key>
  <string>YOUR_APP_KEY</string>
  <key>AccessKeySecret</key>
  <string>YOUR_APP_SECRET</string>
</dict>
```

Note: The credentials are not tied to the name of your app, but to the account used to obtain the credentials. This makes it possible to use the same set of credentials for multiple apps.

Adding the Map View

Now that we have the SDK integrated into the project and added the required credentials, we can add a new map view. There are two alternative ways to do this. We will first show how to add a `MapView` programmatically, then we will show how to add a `MapView` using a Storyboard.

Either way, only one `MapView` instance can be created at a time.

Whichever method you choose, you will need to add the following line to your `ViewController.swift` file:

```
import heresdk
```

Adding the Map View programmatically

Add a map view variable by adding the following variable declaration to your `ViewController` class:

```
var mapView: MapView!
```

In your `viewDidLoad()` method, add the following code to initialize the map view:

```
override func viewDidLoad() {  
    super.viewDidLoad()  
  
    // Initialize MapView without a storyboard.  
    mapView = MapView(frame: view.bounds)  
    view.addSubview(mapView)  
}
```

This gives the `MapView` the same size as the parent view and then adds it as a subview.

Adding the Map View to a storyboard

You might prefer not to initialize the `MapView` programmatically as shown above, but rather to build your UI from Xcode's Interface Builder using storyboards. In this case, you don't need to add the above lines to your view controller.

Instead, open your `Main.storyboard` and select the main view (nested under `ViewController`). In the *Identity Inspector*, under `Custom Class`, type `MapView`. For *Module*, type `heresdk`.

Now, open the Assistant editor, control-click the map view and drag it to your `ViewController` to create an outlet. The outlet window will pop up; for the name, type `mapView` and click `connect`. This will add the following line to your view controller:

```
@IBOutlet var mapView: MapView!
```

Now you have an `IBOutlet` accessible from your view controller which you can start to use. Please note that this procedure does not contain any SDK specific actions. `MapView` behaves exactly like any other `UIView`.

Loading a Map Scene

For this first app using the HERE SDK, we want to load one of the default map styles the SDK is shipped with.

You can also create your own fully customized map styles. HERE provides a WYSIWYG [map style editor](#) to allow you to define your own map style.

In our `viewDidLoad()` method, we add the following code to load the scene with a map style representing a `normalDay` map render style:

```
// Load the map scene using a map style to render the map with.  
mapView.mapScene.LoadScene(mapStyle: .normalDay) { (sceneError) in
```

```
if let error = sceneError {
    print("Error: Map scene not loaded, \(error)")
} else {
    // Configure the map.
    self.mapView.camera.setTarget(GeoCoordinates(latitude: 52.518043, longitude: 13.405991))
    self.mapView.camera.setZoomLevel(13)
}
}
```

Note that the `LoadSceneCallback` is called on the main thread when loading the scene is done. `SceneError` values other than `nil` will indicate what went wrong.

From `MapView`, you can access the `Camera` to set some custom map parameters like the location where you want the map centered on, and a zoom level:

```
self.mapView.camera.setTarget(GeoCoordinates(latitude: 52.518043, longitude: 13.405991))
self.mapView.camera.setZoomLevel(13)
```

Note: You can setup the `Camera` as soon as you have a `MapView` instance available. However, you can see the changes only taking effect after the `LoadSceneCallback` has finished.

As an exercise, in the above example you can try to replace the `main` resource string with the following: `satellite`. What do you get?

Lastly, it is a good practice to handle the `didReceiveMemoryWarning()` of your view controller. The SDK provides the `handleLowMemory()` method to safely clean up any unused resources when it is requested by the OS:

```
override func didReceiveMemoryWarning() {
    super.didReceiveMemoryWarning()
    mapView.handleLowMemory()
}
```

Now, it's time to build and run the app. If all goes well, you should see a HERE map covering the whole area of the screen. Congratulations, you have just mastered your first steps in using the brand new HERE SDK for iOS.



Screenshot: Showing main map style.



Screenshot: Showing satellite map style.

Prepare the framework for distribution

As described [above](#), the HERE SDK framework is a *fat binary*. Before you can distribute your app via Apple's App Store, you must remove the included `x86_64` architecture from the HERE SDK framework. To do so, please execute the following [lipo](#) command from your terminal:

```
lipo YOUR_PATH/heresdk.framework/heresdk -remove x86_64 -output YOUR_PATH/heresdk.framework/heresdk
```

Please adapt the path to where you have stored the HERE SDK binary. To verify that it worked as expected, execute:

```
lipo -info YOUR_PATH/heresdk.framework/heresdk
```

You should see: Architectures in the fat file are: `arm64` .

Now, the HERE SDK binary is ready to be used for the distribution of your app. As it is the practice, in Xcode you have to select a *development team* to sign your app, select a *Generic iOS Device* and select *Product -> Archive*.

Note: The steps above are only needed when you plan to distribute your app to others, for example, to publish it on Apple's App Store. During the development phase of your app, this is not needed: You can use the unmodified framework to directly deploy your app to an attached iOS device or to run it on an iOS simulator.

Troubleshooting

When you run into trouble, please make sure to first check the [minimum requirements](#) and the [supported devices](#).

- **I see only a blank white map:** Please make sure that your HERE credentials are valid and set as described in the *Get Started* section above. Also, please make sure that your device is able to make an internet connection. With slower internet connections, it may take a while until you see the first map tiles loaded.
- **Xcode complains that `Mapview` is not known:** Please make sure to integrate the HERE SDK framework as described in the *Get Started* section above.
- **I cannot archive my app for distribution:** Please make sure to split the framework binary as described in the *Get Started* section above.

General Concepts

In the following use case sections, we will guide you through the most common usage scenarios and reveal tips and easy-to-understand guidelines to help you get the most out of the HERE SDK for iOS.

How to use this guide?

You can read this guide in any order. All sections are independent from each other, making it easy to skip any section and to dive straight into the topics which you are most interested in.

- In the [example section](#), you can find the example apps accompanying this user guide.
- If you are interested in building your first app showing a HERE map, please have a look at the [Get Started](#) guide to easily guide you through the first simple steps.

Engines

The HERE SDK contains several engines to execute specific tasks such as finding your location with the `LocationEngine` or requesting search results via the `SearchEngine`. There are many more engines you can use with the HERE SDK and you can read more about them in the dedicated chapters below. However, most engines share common concepts that makes it easier to use them. For example:

- All engines execute their tasks asynchronously and receive their results on the main thread.
- All engines share similar interfaces, callbacks and error handling.
- It is possible to start multiple instances of an engine in parallel.
- An online connection is required (except for the `LocationEngine`).

Set HERE credentials from Property List or programmatically

Usually, engines can operate independently from each other, and require HERE credentials to request data. The credentials can be set in the `Info.plist` file as shown in the [Get Started](#) guide - or programmatically. For this purpose, you need to create your own instance of the `SDKNativeEngine`, which can then be used to set or to change your HERE SDK credentials at runtime:

```
let sdkOptions = SDKOptions(accessKeyId: "YOUR_APP_KEY",
                             accessKeySecret: "YOUR_APP_SECRET",
                             cachePath: "")
let sdkNativeEngine = SDKNativeEngine(options: sdkOptions)

do {
    try trafficEngine = TrafficEngine(withSdkEngine: sdkNativeEngine)
} catch let engineInstantiationError {
    fatalError("Failed to initialize traffic engine. Cause: \(engineInstantiationError)")
}
```

This can be useful to, for example, inject credentials from a web service. Please note that you must keep the credential keys/value pairs in your `Info.plist` file, even if you set your credentials programmatically - as shown above. However, if you set your credentials programmatically, you can leave the key and secret values empty in the property list.

If you don't set your credentials programmatically, the HERE SDK will be initialized automatically using the values found in the `Info.plist`. Either way, invalid credentials will not block execution until these credentials are used to authenticate your app when you start to use an engine to request data.

Use engines with or without a MapView

It is possible to run an engine as a *stand-alone*, which means that you do not need to add a `MapView` to your application. This way, you can build an app solely around a specific engine. If you are using an engine as a stand-alone, you must take care to initialize the HERE SDK yourself:

```
do {
    try SDKInitializer.initializeIfNecessary()
} catch {
    fatalError("Failed to initialize HERE SDK. Cause: \(error)")
}
```

A good place to do this is the `AppDelegate` after application launch. Then you can proceed to create a new engine in your `ViewController`:

```
do {
    try trafficEngine = TrafficEngine()
} catch let engineInstantiationError {
    fatalError("Failed to initialize traffic engine. Cause: \(engineInstantiationError)")
}
```

When you use the default constructor to initialize an engine for stand-alone usage, the HERE SDK will use a shared `SDKNativeEngine` under the hood to take the credentials as found in the `Info.plist` file. Alternatively, you can provide the credentials programmatically as shown in the previous section.

Custom Map Styles

The HERE SDK comes already with a set of pre-configured map styles such as `MapStyle.NORMAL_DAY` or `MapStyle.SATELLITE`. However, it is also possible to create your own map styles using a configuration file written in the YAML markup format. Such files can be generated with a WYSIWYG [map style editor](#) provided by HERE. For login, please use your HERE credentials (AccessKeyId / AccessKeySecret).

The following example shows a simple map style that defines how to render a water layer provided by the HERE Vector Tile API. For more details on the available layers within the [Vector Tile API](#), please refer to the [Layers](#) section of the Vector Tile API documentation.

```
# Global description of the map. In this example
# the map background color is white.
sources:
  omv:
    type: OMV
scene:
  background:
    color: 'rgb(255, 255, 255)'

# This section contains the style information for the layers
# that are present on the map.
layers:
  # User defined name of the rendering layer.
  water_areas:
    # This section defines where the rendering layer takes
    # its data from. 'source: omv' is mandatory for the Vector Tile API.
    # 'layer: water' specifies what vector layer is taken for rendering.
    data:
      source: omv
      layer: water
    # This section defines how to render the layer.
    draw:
      polygons:
        order: 1 # z-order of the layer
        color: 'rgb(153, 206, 255)'
```

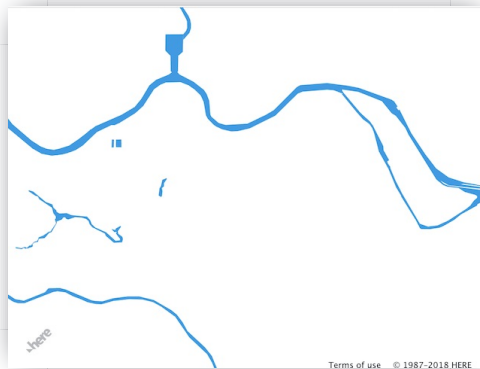
The map style above defines the map scene's background color and a style named `water_areas` that defines how to render the OMV data from the Vector Tile API's `water` layer.

You can paste this custom map style into the [map style editor](#) and tweak it as much as you like. As soon as you are satisfied, it can be loaded into a map scene as follows:

```
let bundle = Bundle(for: ViewController.self)
// Adjust file name and path as appropriate for your project.
let yamlResourceUrl = bundle.url(forResource: "myCustomMapStyle",
                                withExtension: "yaml",
                                subdirectory: "MyData/style")
guard let yamlResourceString = yamlResourceUrl?.absoluteString else {
    print("Error: Map style not found!")
    return
}

// Load the map scene using the YAML resource.
mapView.mapScene.loadScene(sceneConfigPath: yamlResourceString) { (loadSceneError) in
    if (loadSceneError != .none) {
        print("Error: Map scene not loaded.")
    }
}
```

Save the YAML description in a text file and adapt the path to your `Bundle` resources. The screenshot below shows the result.



Screenshot: The map after loading a custom map style configuration showing only water.

Using the custom map styles can be very effective to differentiate your app from other map applications. Besides, it is possible to change map styles on-the-fly, for example, to shift the user's attention on certain map elements based on the current state of your app. Please keep in mind that enabling too many layers might cause performance drawbacks. In the same way, you can use adapted map styles to improve performance on low-end devices, for example, by disabling extruded buildings.

Camera

The HERE SDK for iOS provides several ways to change the view of the map. While with map styles you can change the look of the map, you can use a camera to change the perspective of how a user will look at the map.

For example, the HERE SDK for Android allows you to set a target location, tilt the map, zoom in and out or set a bearing.

At a glance

- Use the `Camera` accessible by `mapView.camera` to manipulate the view of the map.
- Call `camera.updateCamera(cameraUpdate:)` to apply all camera properties in one go.
- Monitor changes to a camera by registering a `CameraObserver`.
- Put constraints on a camera by setting limits to the `CameraLimits` object accessible by `camera.limits`.
- Convert between view and geographical coordinate spaces by using `geoToViewCoordinates(geoCoordinates:)` and `viewToGeoCoordinates(viewCoordinates:)`.
- Get the bounds of the currently displayed area by accessing `camera.boundingRect`.

By default, the camera is located centered above the map. From a bird's eye view looking straight-down, the map is oriented *North-up*. This means that on your device, the top edge is pointing to the north of the map.

You can change the orientation of the map by setting a bearing angle. 'Bearing' is a navigational term and it is counted in degrees, clockwise from North.

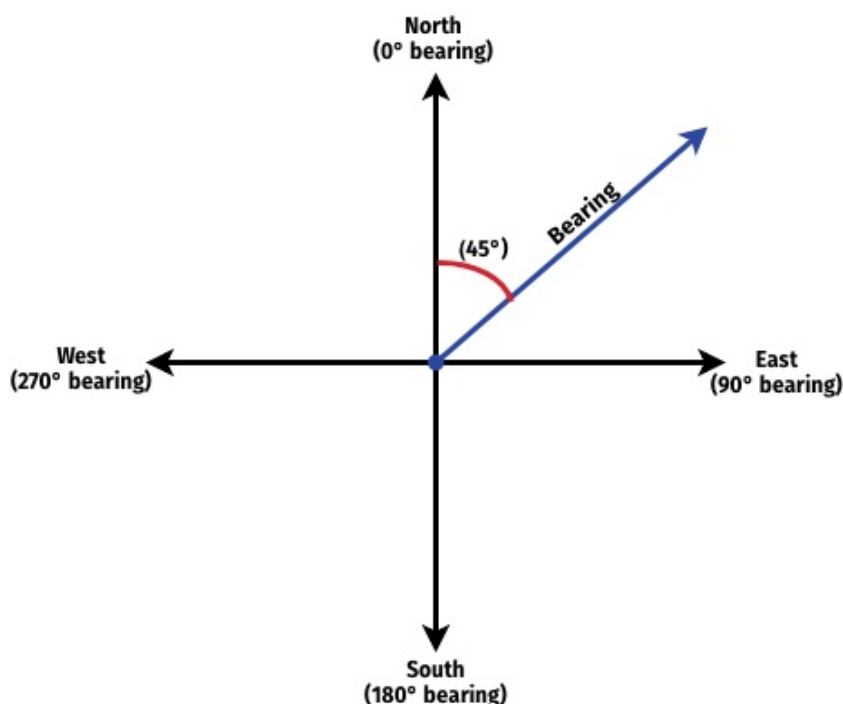


Illustration: Set a bearing direction.

By default, the map is set to a bearing of 0°. When setting an angle of 45°, as visualized in the illustration above, the map will rotate to the *counter-clockwise* and the direction of the bearing will become the new upward direction on your map - pointing to the top edge of your device. This is similar to holding and rotating a paper map while hiking in a certain direction. Apparently, it is easier to orient yourself if the top edge of the map points in the direction in which you want to go. However, this will not always be the true North direction (bearing = 0°).

The camera can also be used to turn the flat 2D map surface to a 3D perspective to see, for example, roads at a greater distance that may appear towards the horizon. By default, the map is not tilted (tilt = 0°).

The tilt angle is calculated from the vertical axis at the target location. This direction pointing straight-down below the observer is called the *nadir*. As visualized in the illustration below, tilting the map by 45° will change the bird's eye view of the camera to a 3D perspective. Although this will effectively move the camera, any subsequent tilt values set will always be applied from the camera's default location. Tilt values above 87° and below 0° are clamped to the range [0, 87], otherwise, no map would be visible. These absolute values are also available as constants, `CameraLimits.minTilt` and `CameraLimits.maxTilt`. You can specify your own limits within this absolute range by setting custom values to `CameraLimits`.

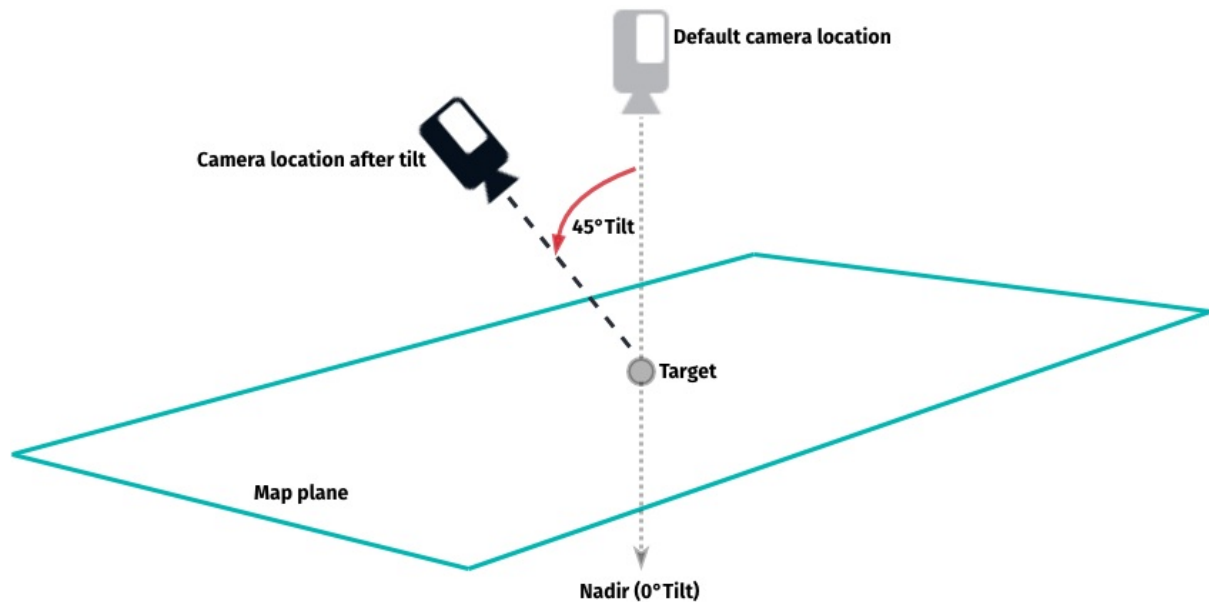


Illustration: Tilt the map.

Using the camera is very simple. See the following code snippets:

```
mapView.camera.setTarget(GeoCoordinates(latitude: 52.530932, longitude: 13.384915))
mapView.camera.setZoomLevel(14)
mapView.camera.setTilt(degreesFromNadir: 60)
```

Alternatively, you can apply multiple changes in one go by setting a `CameraUpdate`.

The camera also allows you to specify a custom range to limit specific values, and it provides a way to observe updates on these values, for example, when a user has interacted with the map by performing a double tap gesture.

By adding an observer and conforming to the `CameraObserver` protocol, your class can be notified when the camera (and therefore the map) is changed:

```
func onCameraUpdated(cameraUpdate: CameraUpdate) {
    let mapCenter = cameraUpdate.target
    print("Current map center location: \(mapCenter). Current zoom level: \(cameraUpdate.zoomLevel)");
}

func addCameraObserver() {
    mapView.camera.addObserver(observer: self)
}
```

In addition, you can manually perform fast and smooth interactions with the map. By default, a double tap zooms in, and panning allows you to move the map around with your fingers. You can read more about default map behaviors in the [Gestures](#) section.

Gestures

As you may have seen from the [Get Started](#) example, a `MapView` by default supports all common map gestures, for example, pinch and double tap to zoom in. The following table is a summary of the available gestures and their corresponding default behavior on the map.

 tap	Tap the screen with one finger. This gesture does not have a predefined map action.
 double tap	To zoom the map in a fixed amount, tap the screen twice with one finger.
 long press	Press and hold one finger to the screen. This gesture does not have a predefined map action.
 pan	To move the map, press and hold one finger to the screen, and move it in any direction.
 2 finger pan	To tilt the map, press and hold two fingers to the screen, and move them in a vertical direction. No behavior is predefined for other directions.
 2 finger tap	To zoom out a fixed amount, tap the screen with two fingers.

To zoom in or out continuously, press and hold two fingers to the screen, and



To zoom in or out continuously, press and hold two fingers to the screen, and increase or decrease the distance between them.

The HERE SDK for iOS provides support for the following gestures:

- Tap: `TapDelegate`
- Double Tap: `DoubleTapDelegate`
- Long Press: `LongPressDelegate`
- Pan: `PanDelegate`
- Two Finger Pan: `TwoFingerPanDelegate`
- Two Finger Tap: `TwoFingerTapDelegate`
- Pinch Rotate: `PinchRotateDelegate`

Each delegate provides a dedicated callback that informs you whenever the user performed an action that could be detected, for example, the beginning or the end of that specific gesture. Usually, you want to add a specific behavior to your application after a gesture was detected, like placing a map marker after a long press.

Note that only one delegate can be set at a time for the same gesture.

Setting a delegate does not affect default behavior of the gestures. That can be controlled independently by calling `Gestures.enableDefaultAction(forGesture:)` and `Gestures.disableDefaultAction(forGesture:)`. By default, all default behaviors, such as zooming in when double tapping the map, are enabled.

When disabling a default map action, you can still listen for the gesture event. This can be useful when you want to turn off the default action of a gesture to implement your own zooming behavior, for example. All gestures - except for tap and long press - provide a default map action. More details can be found in the overview above.

Let's see an example of how a gesture delegate can be attached to the `MapView`. As soon as you set a delegate, it will receive all related events for that gesture via the dedicated callback, which is `onTap()` in case of a `TapDelegate`. The class conforming to this protocol will act as the delegate:

```
func onTap(origin: Point2D) {
    let geoCoordinates = mapView.camera.viewToGeoCoordinates(viewCoordinates: origin)
    print("Tap at: \(String(describing: geoCoordinates))")
}
```

Finally, let the `MapView` know that your class wants to get notified about the tap touch event and start listening:

```
mapView.gestures.tapDelegate = self
```

As soon as you set a delegate, you will begin receiving notifications that gestures have been detected.

The `origin` specifies the point from the device's screen where the gesture happened. By calling `mapView.camera.viewToGeoCoordinates(viewCoordinates: origin)`, you can convert the pixels into geographic coordinates (as shown above).

Likewise, to stop listening, we can simply call:

```
mapView.gestures.tapDelegate = nil
```

For continuous gestures (like long press, pinch, pan, two finger pan), the `begin` gesture state will indicate that the gesture was detected. While the finger(s) still touch the display, you may receive `update` states, until the `end` state indicates that a finger has been lifted.

```
func onLongPress(state: GestureState, origin: Point2D) {
    if (state == .begin) {
        let geoCoordinates = mapView.camera.viewToGeoCoordinates(viewCoordinates: origin)
        print("LongPress detected at: \(String(describing: geoCoordinates))")
    }

    if (state == .update) {
        let geoCoordinates = mapView.camera.viewToGeoCoordinates(viewCoordinates: origin)
        print("LongPress update at: \(String(describing: geoCoordinates))")
    }

    if (state == .end) {
        let geoCoordinates = mapView.camera.viewToGeoCoordinates(viewCoordinates: origin)
        print("LongPress finger lifted at: \(String(describing: geoCoordinates))")
    }
}
```

For example, a user may still keep his finger on the screen after a long press event was detected - or even move it around. However, only the `begin` event will mark the point in time, when the long press gesture was detected.

Note that for the non-continuous gestures (like tap, double tap, two finger tap), no `GestureState` is needed to handle the gesture.

Map Items

The HERE SDK for iOS allows you to add several types of items to the map, such as map polylines, polygons, circles, markers and map overlays. They are explained in detail in the sections below.

- **Map polylines:** Non-moveable rendered lines.
- **Map polygons:** Non-moveable rendered filled shapes.
- **Map circles:** Non-moveable rendered filled circles.
- **Map markers:** Images that can be pinned to 'mark' specific spots on the map.
- **Map overlays:** A convenient way to show native `UIView` layouts on the map.

Polylines, polygons and circles, will adjust their size based on the current zoom level, while markers and overlays remain unchanged when zooming.

All map items provide a convenient way to *pick* them from the map. For polylines, polygons, circles and markers, this can be done using the dedicated `PickMapItemsCallback`, while a `MapOverlay` allows native click handling - as it is for any standard iOS `UIView`.

Add Map Polylines

Polylines can be useful to render, for example, a route geometry on the map. They can be created as shown below:

```
func createPolyline() -> MapPolyline {
    var coordinates = [GeoCoordinates]()
    coordinates.append(GeoCoordinates(latitude: 52.53032, longitude: 13.37409))
    coordinates.append(GeoCoordinates(latitude: 52.5309, longitude: 13.3946))
    coordinates.append(GeoCoordinates(latitude: 52.53894, longitude: 13.39194))
    coordinates.append(GeoCoordinates(latitude: 52.54014, longitude: 13.37958))

    let geoPolyline = try! GeoPolyline(vertices: coordinates)
    let mapPolylineStyle = MapPolylineStyle()
    mapPolylineStyle.setWidth(inPixels: 20)
    mapPolylineStyle.setColor(0x00908AA0, encoding: .rgba8888)
    let mapPolyline = MapPolyline(geometry: geoPolyline, style: mapPolylineStyle)

    return mapPolyline
}
```

A `MapPolyline` consists of three elements:

- A list of two or more geographic coordinates that define where to place the polyline on the map.
- A `GeoPolyline` that contains this list of coordinates.
- A `MapPolylineStyle` to define how to visualize the polyline.

Since a geometrical line is defined by two or more points, you need to create an array list, which must contain at least two `GeoCoordinates`. Otherwise, an exception will be thrown. By using the `MapPolylineStyle` class, attributes such as draw order, thickness in pixels and color of the line can be defined. See the screenshot for an example.

Setting a draw order can be useful to define what map item should be rendered topmost. Higher values will be drawn on top of lower values.

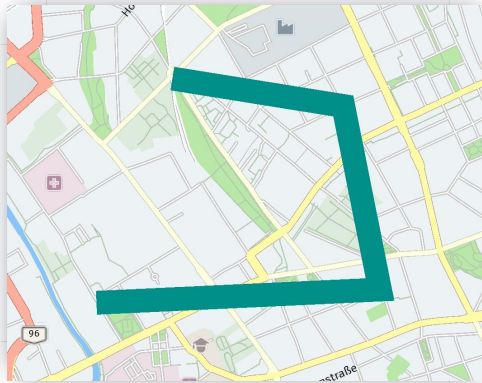
Please note that the chronological order in which the map items are added to the map does *not* influence what item is rendered first. The default draw order value for map items such as `MapPolyline`, `MapPolygon` and `MapCircle` is 1000. The draw order for other map elements, such as streets or rivers, depends on the

current map style you have in use - it can be adapted for custom map styles defined in a YAML file, see the section on [custom map styles](#) for more information.

After you have created one or more map polylines, you can add them to a map scene with:

```
let mapPolyline = createPolyline()
let mapScene = mapView.mapScene
mapView.mapScene.addMapPolyline(mapPolyline)
```

If a map polyline is already attached to a map scene, any further attempt to add it again will be ignored.



Screenshot: Showing a polyline.

Note, a `MapView` allows only one scene and all map items are placed directly on it. If you want to group your map items, you may want to organize them by using an array and add or remove them one by one.

A `mapPolyline` can be immediately removed from the map by calling:

```
mapView.mapScene.removeMapPolyline(mapPolyline)
```

Note: It is possible to store `Metadata` to *pickable* map items such as a `MapPolyline` or a `MapMarker`. For an example, please see the map marker section below.

Add Map Polygons

A `MapPolygon` is a shape that consists of at least three coordinates, otherwise it cannot be rendered. Similar to `MapPolyline`, the coordinates are connected one by one. Polygons can be useful to highlight an area on the map.

Note: The order of the coordinates do matter.

See the example below on how a polyline can be created. The coordinates are connected based on the order in the list. The resulting shape can be filled with a color using a new `MapPolygonStyle` instance:

```
func createMapPolygon() -> MapPolygon {
    var coordinates = [GeoCoordinates]()
    coordinates.append(GeoCoordinates(latitude: 52.53032, longitude: 13.37409))
    coordinates.append(GeoCoordinates(latitude: 52.5309, longitude: 13.3946))
    coordinates.append(GeoCoordinates(latitude: 52.53894, longitude: 13.39194))
    coordinates.append(GeoCoordinates(latitude: 52.54014, longitude: 13.37958))

    let geoPolygon = try! GeoPolygon(vertices: coordinates)
    let mapPolygonStyle = MapPolygonStyle()
    mapPolygonStyle.setFillColor(0x00908AA0, encoding: .rgba8888)
    let mapPolygon = MapPolygon(geometry: geoPolygon, style: mapPolygonStyle)

    return mapPolygon
}
```

```
}
```

A `MapPolygon` consists of three elements:

- A list of three or more geographic coordinates that define where to place the polygon on the map.
- A `GeoPolygon` that contains this list of coordinates.
- A `MapPolygonStyle` to define how to visualize the polygon.

Since a polygon is defined by three or more points, you need to create an array list, which must contain at least three `GeoCoordinates`. Otherwise, an exception will be thrown. By using the `MapPolygonStyle` class, attributes such as fill color and draw order can be defined. See the screenshot for an example.

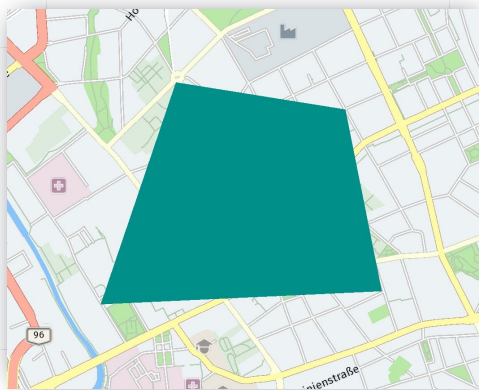
Setting a draw order can be useful to define what map item should be rendered topmost. Higher values will be drawn on top of lower values.

Please note that the chronological order in which the map items are added to the map does *not* influence what item is rendered first. The default draw order value for map items such as `MapPolyline`, `MapPolygon` and `MapCircle` is 1000. The draw order for other map elements, such as streets or rivers, depends on the current map style you have in use - it can be adapted for custom map styles defined in a YAML file, see the section on [custom map styles](#) for more information.

After you have created one or more map polygons, you can add them to a map scene with:

```
let mapPolygon = createMapPolygon()  
let mapScene = mapView.mapScene  
mapScene.addMapPolygon(mapPolygon)
```

If a map polygon is already attached to a map scene, any further attempt to add it again will be ignored.



Screenshot: Showing a polygon.

Note, a `MapView` allows only one scene and all map items are placed directly on it. If you want to group your map items, you may want to organize them by using an array and add or remove them one by one.

A `mapPolygon` can be immediately removed from the map by calling:

```
mapScene.removeMapPolygon(mapPolygon)
```

Note: It is possible to store `Metadata` to *pickable* map items such as a `MapPolygon` or a `MapMarker`. For an example, please see the map marker section below.

Add Map Circles

A `MapCircle` is a circular shape that can be useful to highlight areas on the map - or to mark a distinct spot on the map.

```
func createMapCircle() -> MapCircle {
    let geoCircle = GeoCircle(center: GeoCoordinates(latitude: 52.530932, longitude: 13.384915),
                              radiusInMeters: 300)
    let mapCircleStyle = MapCircleStyle()
    mapCircleStyle.setFillColor(0x00908AA0, encoding: .rgba8888)
    let mapCircle = MapCircle(geometry: geoCircle, style: mapCircleStyle)

    return mapCircle
}
```

A `MapCircle` consists of three elements:

- A set of geographic coordinates that defines where to place the circle's center on the map.
- A `GeoCircle` that contains these coordinates and the radius of the circle in meters.
- A `MapCircleStyle` to define how to visualize the circle.

By using the `MapCircleStyle` class, attributes such as fill color and draw order can be defined. See the screenshot for an example.

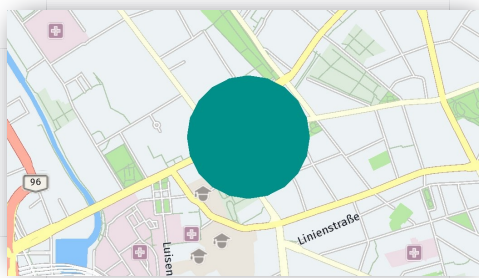
Setting a draw order can be useful to define what map item should be rendered topmost. Higher values will be drawn on top of lower values.

Please note that the chronological order in which the map items are added to the map does *not* influence what item is rendered first. The default draw order value for map items such as `MapPolyline`, `MapPolygon` and `MapCircle` is 1000. The draw order for other map elements, such as streets or rivers, depends on the current map style you have in use - it can be adapted for custom map styles defined in a YAML file, see the section on [custom map styles](#) for more information.

After you have created one or more map circles, you can add them to a map scene with:

```
let mapCircle = createMapCircle()
let mapScene = mapView.mapScene
mapScene.addMapCircle(mapCircle)
```

If a map circle is already attached to a map scene, any further attempt to add it again will be ignored.



Screenshot: Showing a circle.

Note, a `MapView` allows only one scene and all map items are placed directly on it. If you want to group your map items, you may want to organize them by using an array and add or remove them one by one.

A `mapCircle` can be immediately removed from the map by calling:

```
mapScene.removeMapCircle(mapCircle)
```

Note: `MapCircle` items are *pickable*. For an example, please see the map marker section below.

Add Map Markers

You can use map markers to precisely point to a location on the map.

The following method will add a custom map marker to the map:

```
let geoCoordinates = GeoCoordinates(latitude: 52.518043, longitude: 13.405991)
let mapMarker = MapMarker(at: geoCoordinates)

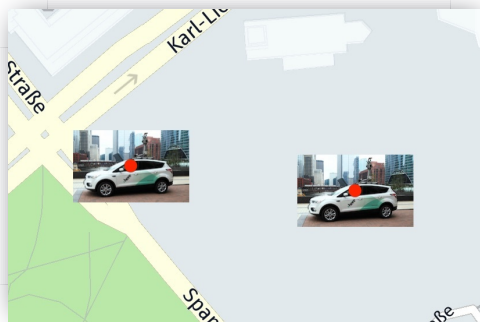
let image = UIImage(named: "here_car.png")
let mapImage = MapImage(image!)
mapMarker.addImage(mapImage!, style: MapMarkerImageStyle())

mapView.mapScene.addMapMarker(mapMarker)
```

In this example, we create a `UIImage` from a PNG ("here_car.png") and convert it to a `MapImage`. This `MapImage` can then be added to a `MapMarker` instance.

The HERE SDK for iOS supports PNG resources with or without transparency (alpha channel) - as well as all other common bitmap resources that are natively supported by `UIImage` including vector graphics such as PDF.

To see the image, we must add the `MapMarker` to a map scene. Please note that the `MapImage` will be displayed centered on the provided `geoCoordinates`.



Screenshot: Showing randomly placed map images. The red map circle on top marks the location of the marker.

You can also update the `geoCoordinates` after the marker is added to the map: It will instantly appear at the new location once the `mapMarker.coordinates` property is updated.

By default, any new `MapMarker` instance is *invisible*, unless we add one or more map images. A `MapMarker` can be composed out of multiple map images - and each map image may use a different anchor point. In the following section below, you can read more about anchor points.

Anchored POI Markers

By default, each image is centered on the provided location. For some marker types, you may want to change that. For example, a POI marker usually points to the location with its bottom-middle position.

Therefore, the location of where the image is rendered must be shifted. The default center lies at (0.5, 0.5). If the bottom-right corner of the view should point to the set `GeoCoordinates` location, then the anchor point must be set to (1, 1).

Anchor points provide a convenient way to specify the location where a marker should be rendered: The top-left corner equals an anchor point of (0, 0) while the bottom-right corner equals an anchor point of (1, 1). Independent of how large the view is, the point that is half of the width or height will always be 0.5 - this is similar to the concept of normalized texture UV-coordinates.

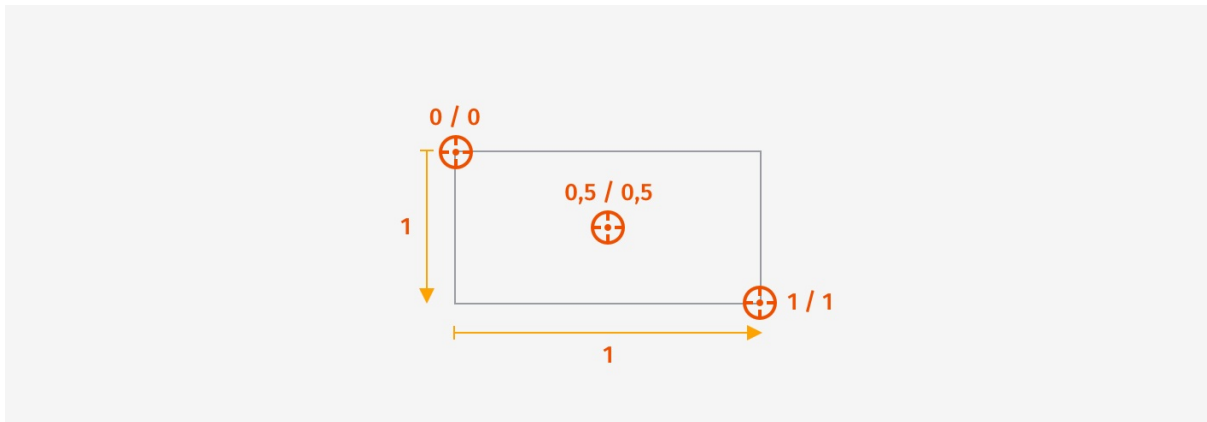


Illustration: Normalized anchor points can specify any location within or outside a map marker.

If you want to shift the POI to point to the location, you can keep the default middle location (0.5), but you must shift the image upwards by 1. 1 is exactly as long as the height of the image. Note that you can also specify values greater than 1 or smaller than 0, so that you can shift the image to any possible location. 2 would represent twice the height of the image and so on.

To add an anchored POI marker to the map, see the example below:

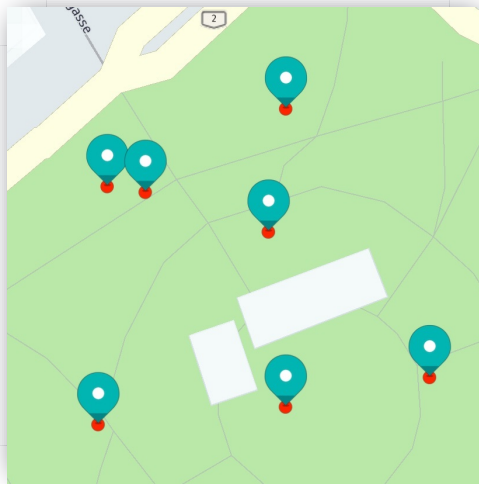
```
let mapMarker = MapMarker(at: geoCoordinates)
let image = UIImage(named: "poi.png")
let mapImage = MapImage(image!)

let mapMarkerImageStyle = MapMarkerImageStyle()
mapMarkerImageStyle.setAnchorPoint(Anchor2D(horizontal: 0.5, vertical: 1))

mapMarker.addImage(mapImage!, style: mapMarkerImageStyle)
mapView.mapScene.addMapMarker(mapMarker)
```

You can add multiple map images to a single map marker. When you move the location of a map marker by updating the `mapMarker.coordinates` property, then all of its added images will move along with it, keeping their relative positions.

Note: For the example, a custom POI image called "poi.png" is added in different resolutions to the project's asset catalog. The iOS platform will choose the appropriate image resolution based on the device's display density. See the accompanying example app on how this can be done.



Screenshot: Showing randomly placed map markers. The red map circle marks the anchored location of the marker.

You can also change the size of a `MapImage` programmatically by setting a scale factor in `MapMarkerImageStyle`. Values below 1 will shrink width/height proportionally, while values greater than 1 will increase the size of the image. The default scale factor is 1, leaving the image unscaled.

Unlike map circles or polylines, each `MapImage` will keep its size - regardless of how much the map is zoomed out or in.

Pick Map Markers

After you have added the map markers onto the map, you can use a tap gesture delegate to find out if a user tapped on a map marker:

```
// Conform to the TapDelegate protocol.
func onTap(origin: Point2D) {
    mapView.pickMapItems(at: origin, radius: 2, callback: self)
}

func onMapItemsPicked(pickedMapItems: PickMapItemsResult?) {
    guard let mapItems = pickedMapItems else {
        return
    }

    guard let topmostMapMarker = mapItems.topmostMarker else {
        return
    }

    showDialog(title: "Map marker picked:", message: "Location: \(topmostMapMarker.coordinates)")
}
```

Our class must then conform to two protocols: `TapDelegate` and `PickMapItemsCallback`. For the first one, we set our class as delegate as usual (`mapView.gestures.tapDelegate = self`). Similarly, for the `PickMapItemsCallback` we set us as delegate each time the user performs a tap gesture.

As soon as the tap gesture is detected, we can use the view coordinates of the tapped location on the screen to ask the map view for any map markers around that location. We can increase the tap radius, but in most cases it is enough to specify a radius of two pixels. The `PickMapItemsCallback` then provides access to the found map items, such as a `MapCircle`, a `MapPolyline`, a `MapPolygon` or a `MapMarker`.

When picking items of the same type, you can compare the instances using the `===` -operator.

Adding Metadata

In many cases, users may want to interact with the shown markers - for example, by tapping on a search result to see more details about a restaurant. For this purpose, a `MapMarker` can hold an instance of the `Metadata` class, so it is possible to attach various types of data to it - even custom types are supported.

`Metadata` can hold several key/value pairs. Below, we create a new key named `"key_poi"` and set a `String` as value containing the information about the type of the marker:

```
let metadata = Metadata()
metadata.setString(key: "key_poi", value: "This is a POI.")
mapMarker.metadata = metadata
```

Certainly, you can set any information you may need. The moment we want to read the contents of a `Metadata` instance, we simply ask for the data stored for a key, which is `"key_poi"` in our example:

```
if let metadata = topmostMapMarker.metadata {
    var message = "No message found."
    if let string = metadata.getString(key: "key_poi") {
        message = string
    }
    showDialog(title: "Map Marker picked", message: message);
    return
}
```

A `MapMarker` instance, by default, does not contain `Metadata` and `mapMarker.metadata` may be `nil`. The data accessed by a key can be `nil` as well, if the `Metadata` object does not contain such information.

If it does, we look for the `String` stored for our key `"key_poi"` and call a helper method to present the contained `String` to the user. You can choose any string as a key based on your preference, but use a unique key, or you will otherwise overwrite the content stored for a different data item. To see the full example's source code, please check the *MapMarker example app*.

You can also store custom objects into `Metadata` using the `CustomMetadataValue` interface. An example can be found in the [Search](#) section where a search result data object is stored as a whole.

Add Map Overlays

While map markers offer a seamless way to place items on the map, map overlays can be used to pin a native `UIView` to the map. This can be useful to show information bubbles, annotations, or customized controls. For example, when tapping on a map marker you can show a map overlay with additional information about the marker location.

Map overlays can hold multiple `UIViews`, and each of them can be used just like any other `UIView`, following user-defined Auto-Layout constraints, animations, and even native gesture handling, allowing the use of, for example, `UITapGestureRecognizer`.

As is the custom, you can define your views in an *Interface Builder* layout file and connect them to your `ViewController` - or generate the desired view content programmatically. For example:

```
let textView = UITextView(frame: CGRect(x: 0, y: 0, width: 200, height: 40))
textView.textAlignment = NSTextAlignment.center
textView.isEditable = false
textView.backgroundColor = UIColor(red: 72/255, green: 218/255, blue: 208/255, alpha: 1)
textView.textColor = UIColor.white
textView.font = UIFont.systemFont(ofSize: 17)
```

```
textView.text = "Centered MapOverlay"
```

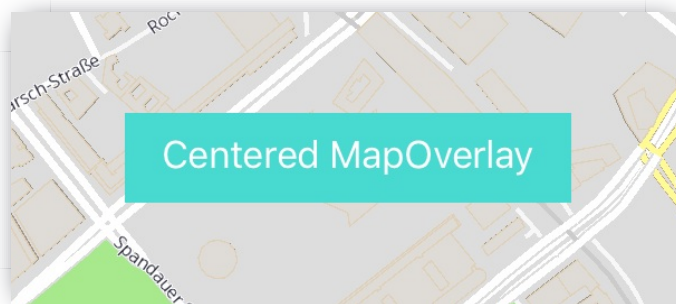
The `UITextView` is a descendant of `UIView`, it can be used directly as an overlay. All we need to do is to add this to a new `MapOverlay` instance and show it on the map:

```
let mapOverlay = MapOverlay(view: textView, geoCoordinates: geoCoordinates)
mapView.addMapOverlay(mapOverlay)
```

Once added to the `MapView` instance, the view - including all nested sub-views - will appear centered on the provided `geoCoordinates` location and its location will automatically adjust to the given geographical location when the map is moved.

Please note that map overlays are always drawn on top of other map items.

Try to remove the line starting with `textView.isEditable`. What will happen? Since the `MapOverlay` makes sure that the *real* view is rendered on the provided location, this will enable the default behavior of an `UITextView` and therefore become editable.



Screenshot: A map overlay showing an UITextView. By default, the view is centered on the provided location.

It is also possible to apply animations to the view or attach some other interaction listeners. Once a view is attached to a `MapOverlay` object, it will behave like any other iOS `UIView` - except that it stays fixed on the map and moves along with it when panning or zooming. Note, the `MapOverlay` does not rotate when you rotate the map - similar to map polylines and map markers.

You can add as many map overlays as you like, but you should take performance into consideration. For example, if you want to indicate multiple search results on the map, then map overlays are less performant than map markers.

To remove an overlay from the map, simply call:

```
mapView.removeMapOverlay(mapOverlay)
```

If you have added multiple `MapOverlay` instances, you can access all overlays from the `MapView` instance by calling:

```
let mapOverlays = mapView.overlays
```

Usually, map overlays are the best choice for showing additional *dynamic* content for a specific location on the map.

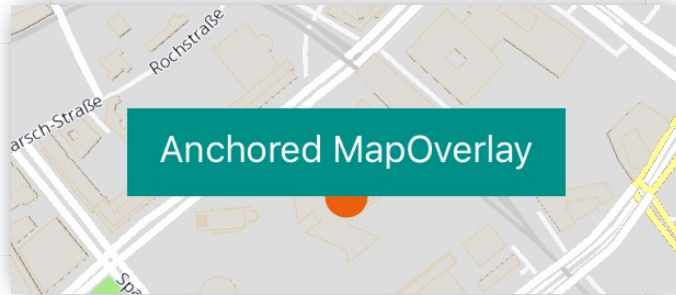
Anchored Map Overlays

By default, the map overlay will be centered on the provided location. But what can you do if you want to use an overlay without covering the area underneath?

For this purpose, you can specify an anchor point - just like you do for any `UIView` :

```
textView.layer.anchorPoint = CGPoint(x: 0.5, y: 1)
```

This will place the view centered horizontally on the location, while the bottom of the view is matching the provided coordinate. As visualized in the screenshot below, the map overlay sits on a map circle object that indicates the provided center location of the map overlay.



Screenshot: An anchored map overlay.

Anchor points provide a convenient way to specify the location where the view should be rendered: Independent of how large the view is, the point that is half of the width or height will always be 0.5 - this is similar to the concept of normalized texture UV-coordinates.

By default, the anchor point is (0.5, 0.5), which will render the view centered on the location. Since a view can be of any size, the maximum width and height will have a value of 1. The dimension of the view is calculated after it is fully inflated. If you know the exact dimensions of your view, you can easily calculate a specific point inside the view in relation to the maximum value of 1.

Note: While *offsets* allows one to specify the translation along the x- and y-axis, an *anchor point* defines the relative position of the top-left corner of a rectangle such as a view. In relation to the view's boundaries, it defines the point where the view is centered on the provided `GeoCoordinates` location.

Search

Sourced from HERE's global data set of hundreds of millions of POIs and Point Addresses worldwide, the HERE SDK for iOS makes it fast and easy to search. The HERE SDK provides a variety of engines tailored for specific demands:

- `SearchEngine` : Search and discover places from HERE's vast database worldwide.
- `ReverseGeocodingEngine` : Search for the address that belongs to a certain geo-coordinate (reverse geocoding).
- `GeocodingEngine` : Search for the locations that belong to an address (geocoding).

One thing that search variants have in common is that you must specify the area where you want to search, except for the `ReverseGeocodingEngine` . Any potential search result that lies outside the specified area is ranked with lower priority, except for relevant global results - for example, when searching for "Manhattan" in Berlin. The underlying search algorithms are optimized to help narrow down the list of results to provide faster and more meaningful results to the user.

In addition, the HERE SDK for Android allows searching for nearby and remote traffic incidents via a dedicated `TrafficEngine` that shares similar interfaces and concepts. The concept is described in more detail [here](#).

Note: Each engine performs the search request asynchronously. An online connection is required.

Places Search

The massive database of places provided by HERE's Location Services can easily be discovered with the HERE SDK's `SearchEngine` . Let's see an example. We begin by creating a new `SearchEngine` instance:

```
do {  
    try searchEngine = SearchEngine()  
} catch let engineInstantiationError {  
    fatalError("Failed to initialize SearchEngine. Cause: \(engineInstantiationError)")  
}
```

Let's assume we want to find all "restaurants" within the current map viewport shown on the device. Before we can start to search, we need to specify some more details:

```
let searchOptions = SearchOptions(  
    languageCode: LanguageCode.enUS,  
    maxItems: 30)
```

From the code snippet above, we created a new `SearchOptions` object that holds the desired data:

- We can specify the language of the returned search results by setting a `LanguageCode` .
- `maxItems` is set to define the maximum number of result items that should be delivered in the response, so we limit the results to 30. If the engine finds more search results than requested, only the 30 most meaningful search results will be returned.

Alternatively, you can use the default constructor `SearchOptions()` to keep the default search options, which are English (US) and 20 search results.

The results which lie near the specified map center are most preferably to be returned. If no results were found, then global search results may be returned instead. Additionally, relevant global results, such as prominent cities or states may be included - regardless of the specified search location.

Note: Along with the search area and the search options, the engine also requires a query string. This string can contain any textual description of the content you want to search for. You can pass in several search terms to narrow down the search results - with or without comma separation. So, "Pizza Chausseestraße" and "Pizza, Chausseestraße" will both lead to the same results and will find only pizza restaurants that lie on the street 'Chausseestraße'.

Let's see an example.

```
searchEngine.search(at: mapView.camera.getTarget(),
    query: queryString,
    options: searchOptions) { (searchError, searchItems) in

    if let error = searchError {
        self.showDialog(title: "Search", message: "Error: \$(error)")
        return
    }

    if searchItems!.isEmpty {
        self.showDialog(title: "Search", message: "No results found")
        return
    }

    self.showDialog(title: "Search", message: "Found \$(searchItems!.count) results.")

    for searchItem in searchItems! {
        // ...
    }
}
```

Before we can look into the results, we should check for a possible `SearchError`. For example, if the device is offline, the list of search items will be nil and the error enum will indicate the cause. In this case, we call a helper method `showDialog()` to show the error description to the user. A possible implementation of `showDialog()` can be accessed from the accompanying "Search" example's source code - it does not contain any HERE SDK specific code.

Note that we can safely unwrap the search items list, as we opt out beforehand if an error occurs.

The search response contains either an error or a result: `searchError` and `searchItems` can never be nil at the same time - or non-nil at the same time.

Now, it's time to look into the results. If no matching results could be found, the list will be empty:

```
if searchItems.isEmpty {
    return
}

for searchItem in searchItems! {
    let searchItemMetadata = SearchItemMetadata()
    searchItemMetadata.searchItem = searchItem
    let metadata = Metadata()
    metadata.setCustomValue(key: "key_search_item", value: searchItemMetadata)
    self.addPoiMapMarker(geoCoordinates: searchItem.coordinates, metadata: metadata)
}

...

// Class to store search items as Metadata.
private class SearchItemMetadata : CustomMetadataValue {

    var searchItem: SearchItem?

    func getTag() -> String {
        return "SearchItem Metadata"
    }
}
```

Finally, we can iterate over the list of results. Each `SearchItem` contains various fields describing the found place.

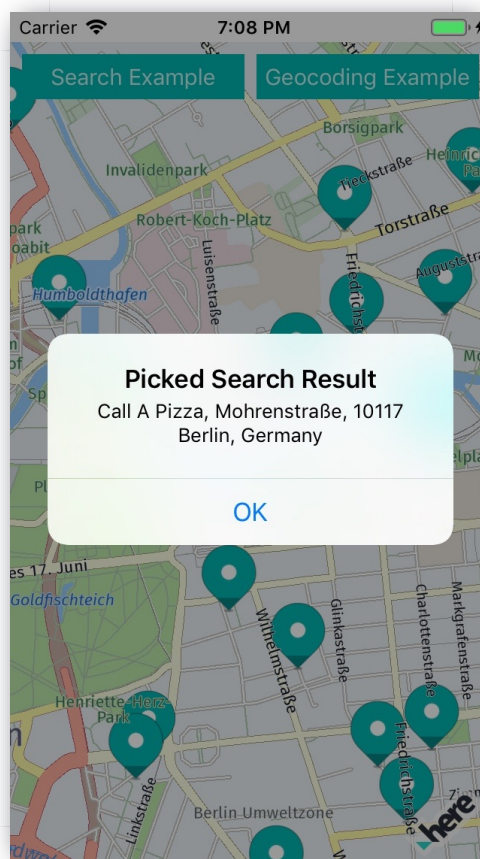
In our example, we are interested in the places's location, in order to add a marker to the map. Additionally, we create a `Metadata` object where we can store a `SearchItem`.

Note: The `Metadata` object can contain various data types to allow easy association of a `MapMarker` with result data. This way, we can hold all information related to a map marker in one object - this can be convenient when presenting this data, for example, after the user taps on a map marker. Even complex data objects can be stored by implementing the `CustomMetadataValue` interface, as shown above.

A possible implementation of `addPoiMapMarker()` can be accessed from the accompanying "Search" example's source code; see also the section about *Map Markers* in this guide. By setting a `TapDelegate`, you can conform to the `PickMapItemsCallback` -protocol to pick a map marker; see the "Search" example's source code for more details - or the section about *Gestures* in this guide. After you have the picked `MapMarker` object at hand, you can get the `Metadata` information which we have set in the previous step:

```
if let metadata = topmostMapMarker.metadata {
    if let customMetadataValue = metadata.getCustomValue(key: "key_search_item") {
        if let searchItemMetadata = customMetadataValue as? SearchItemMetadata {
            if let title = searchItemMetadata.searchItem?.title,
                let vicinity = searchItemMetadata.searchItem?.vicinity {
                showDialog(title: "Picked Search Result",
                           message: title + ", " + vicinity)
                return
            }
        }
    }
}
```

Please note that not all `MapMarkers` may contain `Metadata`. Unless you have set the `Metadata` beforehand, `getMetadata()` will return `nil`. In this example, we simply check if the data stored for `"key_search_item"` is not `nil`, so that we know it must contain search data. We can then downcast to our custom type `SearchItemMetadata` which holds the desired `SearchItem`.



Screenshot: Showing a picked search result with title and vicinity.

Reverse Geocoding an Address from Coordinates

Now we have seen how to search for places at certain locations or areas on the map. But, what can we do if only a location is known? The most common use case for this might be a user who is doing some actions on the map. For example, a long press gesture will provide us with the latitude and longitude coordinates of the location where the user interacted with the map. Although the user sees the location on the map, we don't know any other attributes like the address information belonging to that location.

This is where reverse geocoding can be helpful. First, let's create a new `ReverseGeocodingEngine` :

```
do {
    try reverseGeocodingEngine = ReverseGeocodingEngine()
} catch let engineInstantiationError {
    fatalError("Failed to initialize ReverseGeocodingEngine. Cause: \(engineInstantiationError)")
}
```

Our location of interest is represented by a `GeoCoordinates` instance, which we might get from a user tapping the map, for example. To demonstrate how to "geocode" that location, see the following method:

```
private func getAddressForCoordinates(geoCoordinates: GeoCoordinates) {
    // By default results are localized in EN_US.
    let reverseGeocodingOptions = ReverseGeocodingOptions(languageCode: LanguageCode.enGb)

    // Search for an address and set the ReverseGeocodingCallback.
    reverseGeocodingEngine.searchAddress(coordinates: geoCoordinates,
                                         options: reverseGeocodingOptions,
                                         callback: self)
}
```

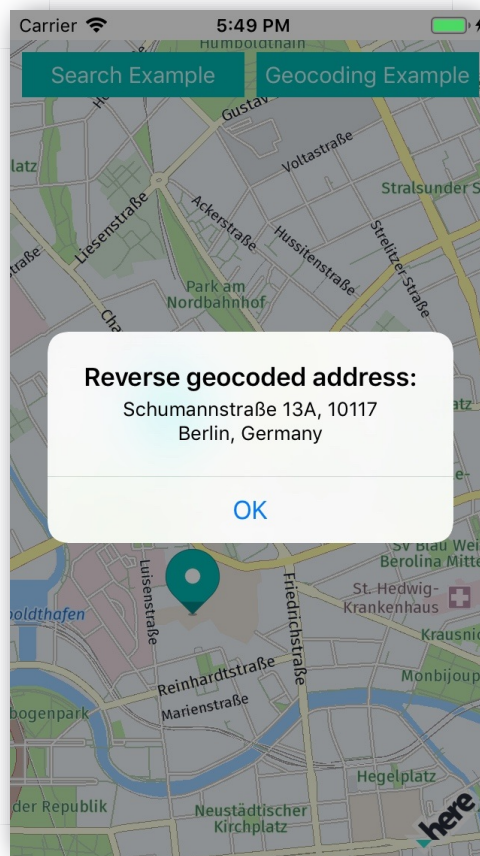
```
// Implement the ReverseGeocodingCallback.
func onSearchCompleted(reverseGeocodingError: ReverseGeocodingError?,
                       address: Address?) {
    if let error = reverseGeocodingError {
        showDialog(title: "ReverseGeocodingError", message: "Error: \(error)")
        return
    }

    let addressText = address!.addressText
    showDialog(title: "Reverse geocoded address:", message: addressText)
}
```

Similar to the other search engines provided by the HERE SDK, a `ReverseGeocodingOptions` object must be provided to set the desired `LanguageCode`. It determines the language of the resulting address. By not specifying a `LanguageCode` - by calling the default constructor - we can request the result in the default language `EN_US`. Then we can make a call to the engine's `searchAddress()` -method to search online for the address of the passed coordinates. In case of errors, such as when the device is offline, `ReverseGeocodingError` holds the error cause.

The reverse geocoding response contains either an error or a result: `ReverseGeocodingError` and `Address` can never be nil at the same time - or non-nil at the same time.

The `Address` object is a data class that contains multiple `String` fields describing the address of the raw location, such as country, city, street name, and many more. Please consult the *API Reference* for more details. If you are only interested in receiving a readable address representation, you can access `addressText`, as shown in the above example. This is a `String` containing the most relevant address details; see the screenshot for an example.



Screenshot: Showing a long press coordinate resolved to an address.

Please note that the `ReverseGeocodingEngine` does not need a certain search area: You can search for addresses worldwide.

Geocoding an Address to a Location

While with reverse geocoding you can get an address from raw coordinates, geocoding does the opposite and lets you search for the raw coordinates and other location details by just passing in an address detail such as a street name or a city.

Note: Whereas reverse geocoding may deliver only one result, geocoding may provide one or many results.

This is how you do it. Similar to using the `ReverseGeocodingEngine`, we first need to create a new `GeocodingEngine` instance:

```
do {
    try geocodingEngine = GeocodingEngine()
} catch let engineInstantiationError {
    fatalError("Failed to initialize GeocodingEngine. Cause: \(engineInstantiationError)")
}
```

Note, we must specify the area where we want to search for an address. In the example below, we specify the current viewport, set a maximum result count of 25 and the `LanguageCode` of the address query:

```
let geocodingOptions = GeocodingOptions(
    languageCode: LanguageCode.deDe,
    maxItems: 25)
```

For this example, we will pass in the street name of HERE's Berlin HQ "Invalidenstraße 116" as the query string. As this is a street name in German, we pass in the language code `deDe` for Germany. Please note that this also determines the language of the returned results.

Alternatively, you can use the default constructor `GeocodingOptions()` to keep the default options, which are English (US) and unlimited results.

Note: Results can lie outside the specified area - although results inside the specified area are ranked higher and are returned preferably.

Now, we can start to asynchronously search for the location details and coordinates that may belong to that address element:

```
geocodingEngine.searchLocations(in: mapView.camera.boundingRect,
                                addressQuery: queryString,
                                options: geocodingOptions,
                                callback: self)
```

Then we must implement the `GeocodingCallback`:

```
func onSearchCompleted(geocodingError: GeocodingError?, items: [GeocodingResult]?) {
    if let error = geocodingError {
        showDialog(title: "Geocoding", message: "Error: \(error)")
        return
    }

    if items?.isEmpty {
        showDialog(title: "Geocoding", message: "No geocoding results found.")
        return
    }

    showDialog(title: "Geocoding", message: "Found: \(items!.count) results.")
}
```

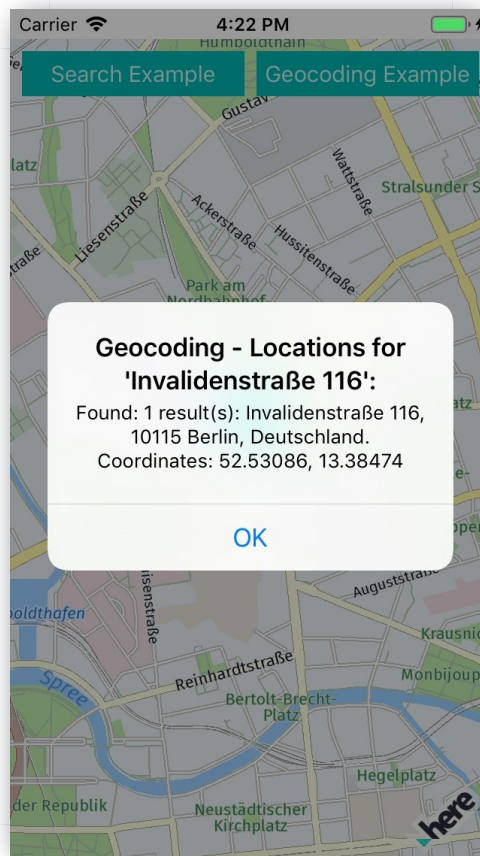
After validating that `GeocodingCallback` completed without an error, we check if the list contains one or more `GeocodingResult` elements.

If `geocodingError` is nil, the result `items` is guaranteed to be not nil, and vice versa. Therefore it's safe to unwrap the optional result list `items`.

Please note that the results are wrapped in a `GeocodingResult` object that contains the raw coordinates - as well as some other address details, such as an `Address` object and the place ID that identifies the location in the [HERE Places API](#) - this is similar to the `ReverseGeocodingEngine` (please see above). Below, we iterate over the list and get the address text and the coordinates:

```
for geocodingResult in items! {  
    if let geoCoordinates = geocodingResult.geoCoordinates, let address = geocodingResult.address {  
        let locationDetails = address.addressText  
            + ". Coordinates: \(geoCoordinates.latitude)"  
            + ", \(geoCoordinates.longitude)"  
        // ...  
    }  
}
```

See the screenshot below for an example of how this can look if the user would pick such a result from the map. If you are interested, have a look at the accompanying "Search" example app that shows how to search for an address text and to place map marker(s) at the found location(s) on the map.



Screenshot: Showing a picked geocoding result.

Traffic

The following chapter describes how to obtain information on traffic incidents and how to visualize them on the map using layer states. In addition, we will show how easy it is to visualize real-time traffic flows on the streets.

Find Nearby and Remote Traffic Incidents

In many situations it may be useful to present accurate and precise information about traffic incidents to the user. The HERE SDK provides a variety of incidents types, as defined in the `IncidentCategory` enum:

- `accident` : An incident caused by one ore more involved vehicles.
- `congestion` : A traffic jam.
- `construction` : Road works.
- `disabledVehicle` : A number of disabled vehicles causing an incident.
- `massTransit` : An incident caused by public transport, such as derailment, injury or police activity.
- `other` : Contains the information about incidents that do not fit into the categories mentioned above.
- `plannedEvent` : An event like a concert, demonstration, movie shot, parade.
- `roadHazard` : Indicates bad road condition, like pothole, traffic light out, downed tree.
- `weather` : A weather condition affecting the road condition, like rain, snow, sleet, wind.
- `roadClosure` : A blocked road.
- `laneRestriction` : One or more blocked lanes.

Each category can have one of four severity attributes, defined in the `IncidentImpact` enum (`critical` , `major` , `minor` , `low`), where `critical` represents the highest possible impact on a driver, and `low` represents the lowest impact respectively.

These incident categories are contained in an `Incident` object, which contains much more useful information.

Let's see how easy it is to access the available traffic incidents information. Firstly, we must create a new

`TrafficEngine` :

```
do {
    try trafficEngine = TrafficEngine()
} catch let engineInstantiationError {
    fatalError("Failed to initialize traffic engine. Cause: \(engineInstantiationError)")
}
```

With this `TrafficEngine` , we can directly request available traffic incidents information in an area specified by a bounding box. This can be the device's viewport or any other area defined by a `GeoBoundingBox` containing North West and South East coordinates. Please note that the maximum size of the bounding box is limited to a size of two degrees per longitude and latitude. The second parameter `incidentQueryOptions` specifies the types of incidents we are interested in:

```
let incidentQueryOptions = IncidentQueryOptions(
    impactFilter: [IncidentImpact.minor],
    categoryFilter: [IncidentCategory.construction],
    languageCode: LanguageCode.enGb)
```

Here we define that we are only interested in minor road works (`construction`). The `IncidentQueryOptions` object accepts two lists defining the impact types of interest and an incident category list. Additionally, we provide a `LanguageCode` to get all information localized to the preferred language: We choose `enGb` to get the results in British English.

When `nil` is passed as language, then the incident's description will be localized based on the country of its origin.

Similarly, you can use the default constructor to turn on all impact types and categories:

```
let incidentQueryOptions = IncidentQueryOptions()
```

Once we have decided which options to set, we can fetch the traffic incidents asynchronously:

```
private func logIncidentsInViewport(incidentQueryOptions: IncidentQueryOptions) {
    trafficEngine.queryForIncidents(in: mapView.camera.boundingRect,
                                   options: incidentQueryOptions) {
        (incidentQueryError, incidents) in

        if let error = incidentQueryError {
            let message = "Query for incidents failed. Cause: \(error)"
            self.showDialog(title: "Error", message: message)
            return
        }

        // ...
    }
}
```

Please note that the `incidents` list is guaranteed to be non-nil when `incidentQueryError` is nil.

To access more detailed information about the found incidents, we can print some information to the console by iterating over the returned incident array:

```
for incident in incidents! {
    print("Incident: \(incident.category)"
          + ", info: \(incident.description)"
          + ", impact: \(incident.impact)"
          + ", from: \(String(describing: incident.startCoordinates))"
          + " to: \(String(describing: incident.endCoordinates))")
}
```

This may produce a log output like shown in the following example:

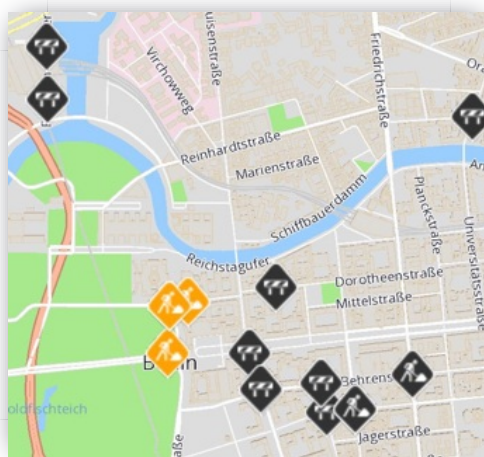
```
Incident: 2, info: between Chausseestraße and Gartenstraße, Road construction. Roadway reduced to one lane, impact: 2, from: Optional(<HERELatLon 0x600002e413a0 latitude:52.53058 longitude:13.38307>) to: Optional(<HERELatLon 0x600002e40c60 latitude:52.531153 longitude:13.385182>)
...
```

Note: All data is fetched online to make sure the engine will always get the most precise and freshest data available. However, this also means that no results will be delivered if the device is offline. In this case,

`IncidentQueryError` would indicate an error *and* the `incidents` list would be nil.

How to show traffic incidents on the map?

You can easily visualize traffic incidents on the map by enabling the `MapLayer` state `trafficIncidents`. The HERE SDK also supports a separate layer to see the current traffic situation. See the traffic flow example below for how to show or hide a layer on the map.



Screenshot: Traffic incidents visualized on the map.

Once a layer is set, the visible area of the map is automatically updated. So you can freely pan the map in all directions to see the latest traffic incidents. Note that you do not need to use the `TrafficEngine` to visualize incidents, since the traffic information is part of the map layer.

Show Real-Time Traffic Flow

In many situations, drivers are interested in finding the fastest route based on the current traffic jams in a city - or outside a city. The HERE SDK allows to show a layer holding all the current traffic jams, visualized by lines in different colors to indicate the severity of the jam - always updated in real-time. This feature requires an online connection and consumes slightly more data. However, the traffic lines are shown as part of the map tiles and are therefore highly performant.

This is how you can show traffic lines on the map by enabling the corresponding layer:

```
mapView.mapScene.setLayerState(layer: MapLayer.trafficFlow, newState: LayerState.enabled, callback: self)

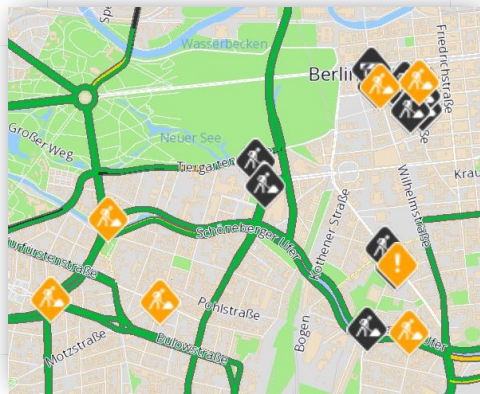
...

// Conforming to SetLayerStateCallback protocol.
func onSetLayerState(sceneError: SceneError?) {
    if let error = sceneError {
        print("Error when setting a new layer state: \(error)")
    }
}
```

Optionally you can set a `SetLayerStateCallback` to get notified on potential errors and the new layer state. For hiding the traffic flow layer, you can call the above with `LayerState.disabled` instead of `LayerState.enabled`.

The traffic flow lines are color coded as follows:

- Green: **Normal** traffic.
- Amber: **High** traffic.
- Red: **Very high** traffic.
- Black: **Blocking** traffic.



Screenshot: Traffic flow visualized on the map together with incidents.

Directions

The HERE SDK provides a full-fledged route engine to calculate the best route directions from A to B, including multiple waypoints and maneuver data for each turn. You can also specify your preferences by setting the desired route type (*fastest* or *shortest*) and various route options (such as speed profiles, route restrictions, vignette options, and more). The HERE SDK supports the following transport modes:

- Online car route directions.
- Online pedestrian route directions.
- Online truck route directions with highly customizable truck options.
- Online bicycle route directions (BETA stage).

Start your trip by creating the route engine in this manner:

```
do {
    try routingEngine = RoutingEngine()
} catch let engineInstantiationError {
    fatalError("Failed to initialize routing engine. Cause: \(engineInstantiationError)")
}
```

As a next step, you can calculate the route based on two waypoints - a starting location and a destination (both of type `Waypoint` that holds a `GeoCoordinates` instance):

```
let carOptions = CarOptions()
routingEngine.calculateRoute(with: [Waypoint(coordinates: startGeoCoordinates!),
                                   Waypoint(coordinates: destinationGeoCoordinates!)],
                           carOptions: carOptions) { (routingError, routes) in

    if let error = routingError {
        self.showDialog(title: "Error while calculating a route:", message: "\(error)")
        return
    }

    let route = routes!.first
    self.showRouteDetails(route: route!)
    self.showRouteOnMap(route: route!)
}
```

You can call `calculateRoute()` multiple times, for example, to calculate routes with different routing options in parallel.

Each route calculation will be performed asynchronously. Once completed, you get a `Route` list or a `RoutingError` that holds a possible error. It is nil if all goes well. In case of an error, the route list is nil. For example, the engine cannot calculate routes when a route would not be feasible for the given mode of transportation.

If there is no error, the route list will contain only *one* result. By specifying the number of route alternatives via the route options, you can request *additional* route variants. By default, a route will be calculated with no route alternatives.

The `showRouteDetails()` -method from the code snippet above is used to show more route details including maneuver data. You can find the full source code in the accompanying example app. Maneuvers are also explained in greater detail below. The `showRouteOnMap()` -method contains an example, how to render a route on the map. We will explain this in a minute below.

Routing Options

For the example above, we have set a new `CarOptions` instance to calculate a car route. You can calculate routes for other transport modes by using dedicated route options for each transport mode.

You can calculate pedestrian, bicycle or even truck routes. Please note that bicycle routing is in a BETA state and may lead to unexpected results. All of these options allow you to further specify several parameters to optimize the route calculation to your needs.

By default, a route will be calculated using the `fastest` route mode.

You can also specify the algorithm that will be used to optimize the route calculation for your needs. For example, choose the `fastest` route mode via `RouteOptions` if you want to reach your destination quickly and the length of the route is less important to you. If you prefer a shorter route and time isn't so important, then choose `shortest`. If you don't want to have such a trade-off between time and length, then you may want to select `balanced`.

The routing algorithm takes many different parameters into account to find the best possible route for you. This does not mean that the algorithm will always provide the absolute *shortest* or *fastest* route. For example, consider the following road network:

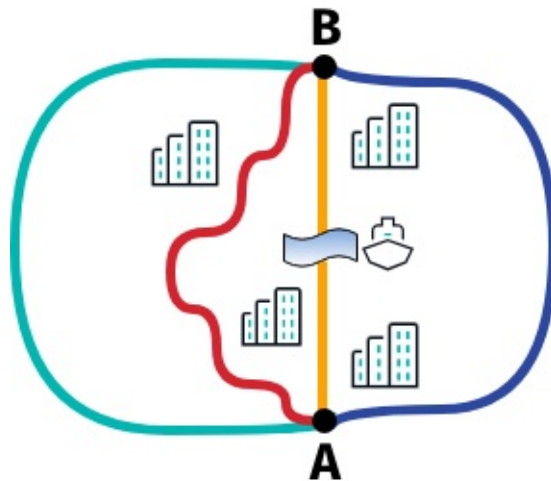


Illustration: Which route would you choose?

When you plan a trip from A to B, you may have the choice between *four* different roads. Let's assume that the green route represents a highway, then this road may be the *fastest* route, although it is longer than any of the other routes that would guide you through a city.

If you prefer to take the *shortest* route instead, then the route algorithm may favor the blue route, although the yellow and the red routes are shorter. Why is that so? Obviously, the yellow road is the absolute shortest route, but it has to cross a river where a ferry must be taken. This might be considered time-costly by the algorithm. Therefore - as a result - the red or the blue route may be favored over the yellow route, although both are slightly longer.

Let's look at the remaining two choices. When comparing the blue and the red route, then the routing algorithm may suggest the blue route as the shortest, although it is slightly longer than the red route: Taking many turns is usually not favorable for a driver. In this case, the red route contains more turns than the blue route, but as it is only slightly shorter, the blue route may be the preferred route. Internally, the routing algorithm penalizes turns and many other road properties, such as traffic lights or rail crossings that may slow down a driver.

Along with the common routing options, the HERE SDK offers specialized options for the various supported transport modes, such as truck routing, where you can specify, for example, the dimensions of your truck to find only the suitable routes where your truck would fit - considering parameters such as road width or tunnel height.

While the resulting routes are optimized based on certain criteria, there may be situations where you don't want to rely on that. Imagine a city trip through Berlin - finding the fastest or shortest route may not be an option if you want to experience many sightseeing spots the city has to offer. In such a case, setting additional waypoints may be a good idea. Please find an example below.

Shape the Route with additional Waypoints

By default, when passing only start and destination waypoints, the resulting route will contain only one route leg. Each route object can contain more route legs depending on the number of set waypoints.

Waypoints are coordinates that will shape the route (which can be useful, if for example, you want to ensure to cross a river by taking a ferry).

There can be two types of waypoints:

- `stopOver` : The default waypoint type. It is guaranteed that this point will be passed, therefore it appears in the list of maneuvers, and splits the route into separate route legs.
- `passThrough` : May not appear in the maneuver list and is rather treated as a hint to shape the route, for example, as a result of a touch input. This type will not split the route into separate legs.

When creating a new `Waypoint` object, the `stopOver` type is set by default - and this must be the type used for the first and the last waypoint. With just two waypoints acting as the start and destination, a route may look like below:

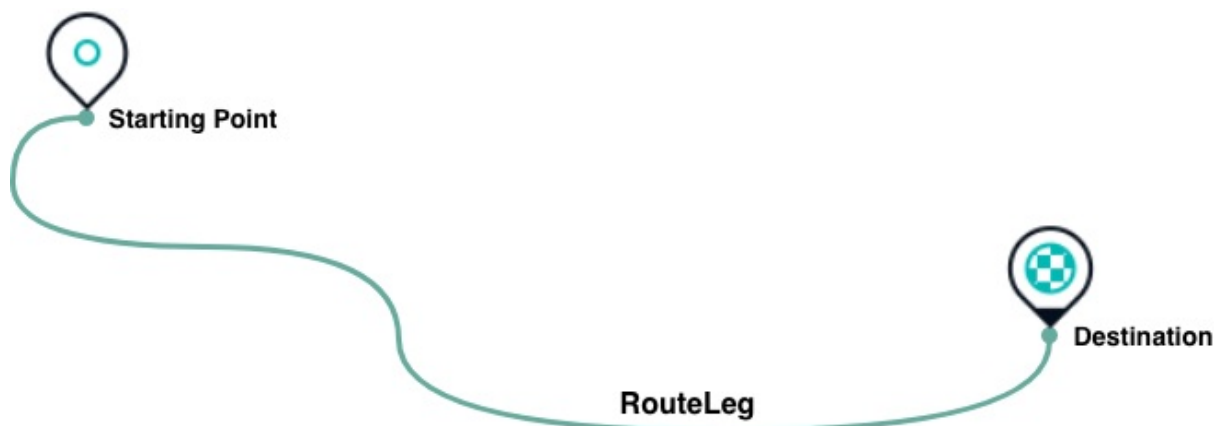


Illustration: One route leg between starting point and destination.

The `RoutingEngine` can handle multiple waypoints. The underlying algorithm will try to find the best path to connect all waypoints while respecting the order of the provided array - as well as the `WaypointType`. Each `stopOver` -waypoint is passed between the starting location and the destination, which are the first and last item of the waypoint list respectively.

```
let waypoint1GeoCoordinates = createRandomGeoCoordinatesInViewport()
let waypoint2GeoCoordinates = createRandomGeoCoordinatesInViewport()
let waypoints = [Waypoint(coordinates: startGeoCoordinates),
                 Waypoint(coordinates: waypoint1GeoCoordinates),
                 Waypoint(coordinates: waypoint2GeoCoordinates),
                 Waypoint(coordinates: destinationGeoCoordinates)]

let carOptions = CarOptions()
routingEngine!.calculateRoute(with: waypoints,
```

```

        carOptions: carOptions) { (routingError, routes) in

        if let error = routingError {
            self.showDialog(title: "Error while calculating a route:", message: "\(error)")
            return
        }

        let route = routes!.first
        self.showRouteDetails(route: route!)
        self.showRouteOnMap(route: route!)
    }

```

By adding two additional `stopOver` -waypoints to the route using the code snippet above, we now have three route legs between the starting point and the destination as seen in the illustration below.

Please keep in mind that the list order of the waypoints defines in which order they will be passed along the route.

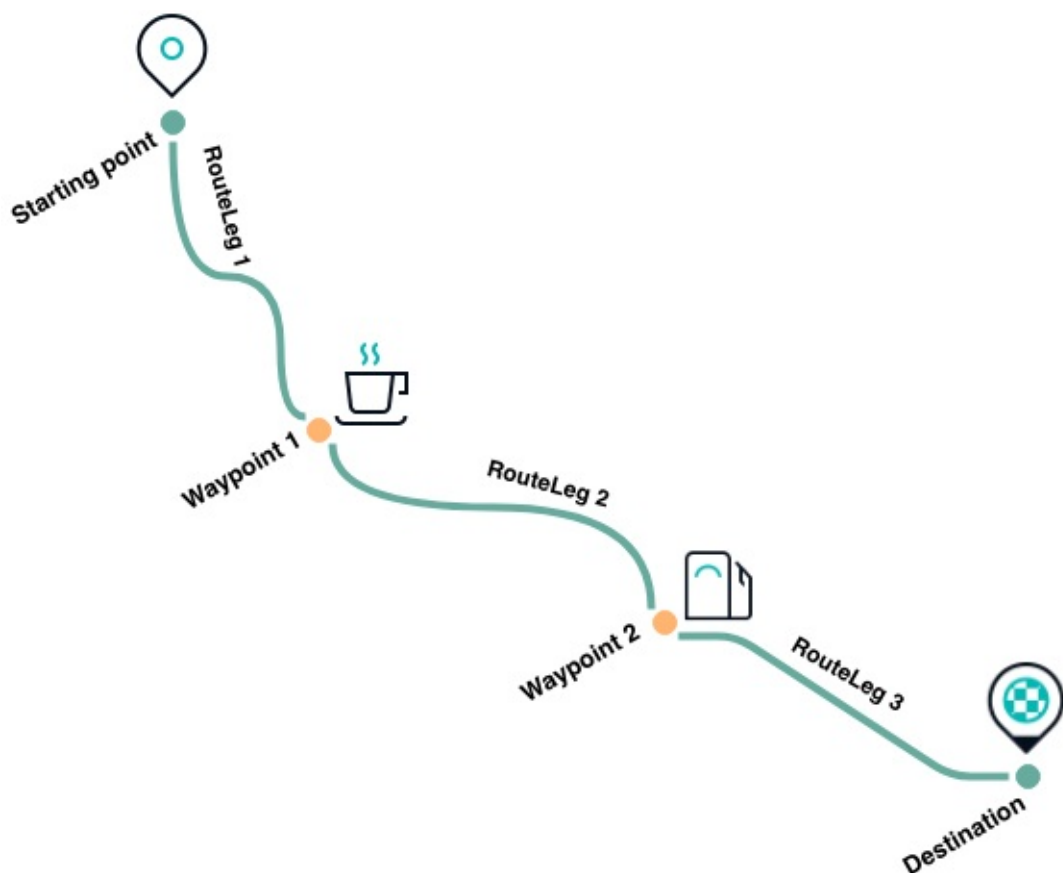


Illustration: Adding two additional waypoints.

Additional information on the route - such as the estimated time it takes to travel to the destination and the total length of the route in meters - can be retrieved from the `Route` object as shown below:

```

let estimatedTravelTimeInSeconds = route.travelTimeInSeconds
let lengthInMeters = route.lengthInMeters

```

Travel time and length are also available for each `RouteLeg`. Without additional `stopOver` -waypoints, the route will contain only one `RouteLeg`. If additional `stopOver` -waypoints are provided, the route is separated into several route legs between each waypoint, as well as from the starting point to the first waypoint and from the last waypoint to the destination.

By default, an additional waypoint splits a route into separate legs and forces the route to pass this point and to generate a maneuver for it.

Each `RouteLeg` contains the shape of the route represented as an array of coordinates where the first coordinate marks the starting point and the last one the destination.

This can be useful to visualize the route on the map by using, for example, map polylines in a different color for each `RouteLeg`. However, you can also get the shape directly from the `Route` object. Below is a code snippet that shows how this could be implemented by using a `MapPolyline` that is drawn between each waypoint including the starting point and the destination:

```
let routeGeoPolyline = try! GeoPolyline(vertices: route.shape)
let mapPolylineStyle = MapPolylineStyle()
mapPolylineStyle.setColor(0x00908AA0, encoding: .rgba8888)
mapPolylineStyle.setWidth(inPixels: 10)
let routeMapPolyline = MapPolyline(geometry: routeGeoPolyline, style: mapPolylineStyle)
mapView.mapScene.addMapPolyline(routeMapPolyline)
```

The first screenshot below shows a route without additional waypoints - and therefore only one route leg. Starting point and destination are indicated by green-circled `MapMarker` objects. Note that the code for drawing the circled objects is not shown here, but can be seen from the example's source code, if you are interested.



Screenshot: Showing a route on the map.

The second screenshot shows the same route as above, but with two additional `stopOver`-waypoints, indicated by red-circled `MapMarker` objects. The route therefore, contains three route legs.



Screenshot: Showing a route with two additional waypoints.

Maneuvers

Each `RouteLeg` contains the maneuver instructions a user may need to follow to reach the destination. For each turn, a `Maneuver` object contains an action, the direction, and the location where the maneuver must be taken.

```
let routeLegs = route.legs
for routeLeg in routeLegs {
    logManeuverInstructions(routeLeg: routeLeg)
}
```

And here is the code to access the maneuver data per leg:

```
private func logManeuverInstructions(routeLeg: RouteLeg) {
    print("Log maneuvers per route leg:")
    let maneuvers = routeLeg.maneuvers
    for maneuver in maneuvers {
        let maneuverAction = maneuver.action
        let maneuverDirection = maneuver.direction
        let maneuverLocation = maneuver.coordinates
        let maneuverInfo = "\(maneuver.instruction)"
            + ", Action: \(maneuverAction)"
            + ", Direction: \(maneuverDirection)"
            + ", Location: \(maneuverLocation)"
        print(maneuverInfo)
    }
}
```

Additionally, an (optionally) HTML-formatted instruction is included. When logging the above `maneuverInfo` to the console, it may produce a result as printed below:

```
Take the <span class="exit">2nd exit</span> from Moritzplatz roundabout onto
<span class="next-street">Prinzenstraße</span> toward
<span class="sign"><span lang="de">Neukölln</span></span>.
<span class="distance-description">Go for <span class="length">692 m</span>.
</span>,
Action: 42,
Direction: 3,
Location: <GeoCoordinates 0x600003c0d120 latitude:52.503866 longitude:13.4109056>
```

This may be useful to easily build maneuver lists describing the whole route in written form. The `ManeuverAction` and `ManeuverDirection` enums can be used to build your own unique routing experience.

Coverage Information

Feature support in the HERE SDK for iOS may differ depending on the language and locale. The following list provides an overview of the currently available coverage:

- [Satellite Imagery](#)
- [Traffic](#)
- [Routing](#)
- [Point Addresses](#) (such as house numbers)
- [Geocoding / Reverse Geocoding](#)
- [Places and Search](#)

You can find more details on our [coverage and availability](#) page for each service we offer within the HERE Location Services and the HERE SDK for iOS.

We are constantly updating our map data and extending our services to new regions, so keep an eye on this page as new content and supported areas will be added soon.

API Reference

The accompanying API Reference is bundled within your downloaded package from our [S3 bucket](#) or can be found on developer.here.com. If not available, please ask your HERE representative.

Examples

The accompanying example apps are bundled within your downloaded package from our [S3 bucket](#) or can be found on developer.here.com. If not available, please ask your HERE representative.

To try out these examples, please add your HERE credentials (`AccessKeyId` and `AccessKeySecret`) to the `Info.plist` file, and add the HERE SDK framework to the app's root folder. For more details on integrating the HERE SDK, please consult the [Get Started](#) section in this *Developer's Guide*.