VMSDK User Guide

Version 1.2 for Android

Contents

Welcome 4
  Audience 5
  Supported platforms 5
Get started 5
  Install and run the reference app 5
Implement 6
  Load the VMD file and display your map 6
    Legacy Support 7
  Customizing your map's look and feel 7
    Global map styling 8
    Styling individual map elements 8
  Responding to MapView events and callbacks 8
    Map loading will start 9
    Map loading complete 9
    Changes in map position 9
    Room selection/highlighting 10
    Adding your own annotations to the map view 10
  Responding to Errors that occur in the SDK 11
    VMD Parsing errors 11
    Map display errors 12
Enable Wayfinding 12
  Handle Wayfinding events 12
  Override auto-generated wayfinding directions and landmark names 13
  Responding to wayfinding events and callbacks 14
    Changing floors for wayfinding 14
  Responding to errors that occur in wayfinding 15

Wayfinding errors 15
Customizing wayfinding look and feel 16
Landmark customization 16
More Information 16
Appendix: Vector Map Tile Style Spec 17
Supported spec versions: 1.0, 1.1* 17
Style spec properties 17
Possible Style Layer ID Patterns 21

Copyright © 2019 Aegir

Welcome

Aegir uses groundbreaking technology to map venue spaces and give your users more choice, convenience, and control. Aegir's **Venue Map Data (VMD)** specification is the foundation, acting as a high-fidelity map and data repository.

VMD is a specification based upon industry standard digital mapping technology that provides a comprehensive geolocated data set for venue maps in which all geometries and points that make up the map shapes and points of interest are accessible, as well as metadata for use with the SDK to support extended applications. Potential use cases might include:

- Wayfinding
- Interior positioning, geofencing applications
- Room/space/unit selection
- Facilities management: environmental controls, housekeeping status and assignments, digital key hardware management, etc

Aegir's **Venue Map Software Development Kit (VMSDK) for Android** provides a suite of functionality for mobile app development based on the VMD specification, including:

- Auto-generated wayfinding paths
- Auto-generated wayfinding directions with support for override from a separate data file

- Support for Vector and Raster map tiling on top of Google and Apple Maps

Audience

VMSDK documentation is designed for people familiar with basic mobile development for Android.

Supported platforms

VMSDK supports Android 5.0+. Consult your map provider's documentation for their own supported platforms.

Get started

With the Venue Maps SDK Reference App, you'll see how to:

- Show a map using a specific map provider, and display your custom map tiles
- Load the VMD file with wayfinding data
- Optionally load additional custom map data for directions and naming
- Allow custom styling for your map tiles
- Allow custom styling for your wayfinding
- Handle errors that occur

Install and run the reference app

1. Extract the Android VMSDK zip file**.**
2. In the extracted directory, navigate to **vmsdk-sampleapp** and open it in Android Studio.
3. Generate a Mapbox Access Token and generate a Google Maps API Key. (The reference app uses Mapbox and Google Maps as example map providers, so you will need to create an account and an access token with Mapbox, and an account and an API Key with Google.)
4. Once you've generated a Mapbox Access Token and Google API Key, open **src/main/res/values/strings.xml** and update **mapbox_access_token** and **google_maps_key** values.
5. Build and run the app on the emulator or device.

NOTE: Using Google Maps with the VMSDK is not supported in some versions of the Android emulator. You will need to use a physical device for this.

Implement

Load the VMD file and display your map

Now you're ready to write code to load your waypoint data and map data into the VMD file. Consult the example in the SDK demo: **MapActivity.java.**

Implement the com.aegir.vmms.vmd.MapDelegate protocol to be notified when the map is finished loading.

You can also load custom map info from a file to override any auto-generated labeling information or to provide additional information where the SDK cannot determine useful points of interest for specific sections of your map.

```
//MapView setup
//Add a VectorMapView in your layout file
<com.aegir.vmms.android.ui.VectorMapView
                android:id="@+id/mapView"
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:visibility="visible"
                vmsdk:mapboxApiKey="@string/mapbox_access_token"
               ></com.aegir.vmms.android.ui.VectorMapView>

//in your Activity's onCreate method, setup your map view
this.mapView = (MapView) findViewById(R.id.mapView);
this.mapView.onCreate(savedInstanceState, this, MAPBOX_SAMPLE_STYLE);


//VMD Setup:
//Configure zip file location
VMDAssetFile zipFile = new VMDAssetFile("venue_map_sample.zip" ,
getApplicationContext().getAssets());

//create a file collection
VMDZipFileCollection zipCollection = new VMDZipFileCollection(zipFile,
getApplicationContext().getCacheDir().getAbsolutePath());

//start loading from the zip file (asynchronously)
Map.load(zipCollection, this, callback);
```

```
...
...
...


//implement MapDelegate interface to get notified when the VMD is done
loading
/**
 * Called when the VMD file is done loading SUCCESSFULLY
   @param vmd a Map object with venue object model and wayfinding data
 */
@Override
public void didFinishLoadingMap(Map vmd, CustomMapInfo customMapInfo)
{
   // this is a good place to create your MapView and call the onCreate()
      method of that MapView instance.
   this.mapView = new VectorMapView(this,
        getResources().getString(R.string.mapbox_access_token));
}
```

At this point, you should have a map that is showing the world with your map tiles superimposed.

Legacy Support

If you need to add support for parsing legacy venue map data files to your application, include the '**vmsdk-legacy.jar**' in your project. The SDK will automatically detect if it's loading a zip file that contains legacy venue map data or new venue map data and process appropriately.

NOTE: Vector map tiles are not supported for legacy venue maps, so you will need to use the RasterMapView class for displaying legacy venues instead of the VectorMapView.

Customizing your map's look and feel

If you use vector map tiles, as opposed to raster map tiles, you have great flexibility in styling your map. This customization also applies to raster map tiles, although it's much more limited. For more information, see **Appendix: Vector map tile style spec** below.

Global map styling

The easiest way to style your map is to create a Map Style JSON configuration file that follows the style spec in the appendix. For a full example, see **style_default.json** in the demo project.

```
//load your custom style from style_default.json configuration file
VMDAssetFile venueStyleConfig = new VMDAssetFile("style_default.json",
getAssets());
VenueStyle venueStyle = new VenueStyle(venueStyleConfig,"venue_map_sample" );

//apply the style to your map
this.mapView.setStyle(venueStyle);
```

Styling individual map elements

You can now add custom styles to individual map elements. This allows you to override the overall map style defined in your map's VMVenueStyle configuration for a single element.

```
//get the map element
MapUnit element = ...
MapView mapView = ...

//create your custom style

VenueLayerStyle style = new VenueLayerStyle();
style.setFillColor(Color.GREEN);
style.setOutlineColor(Color.RED);

//... see class documentation for a full list of styleable attributes

//apply the style to the unit. If the map unit is not visible, it will be
applied the next time it is shown.
mapView.setStyleForMapUnit(style, element);
```

Responding to MapView events and callbacks

There are numerous ways to customize the behavior of your map view by responding to the following events:

Map loading will start

Before you can begin configuring your map view with its initial state, you must wait for internal setup to happen first. After that is complete, your app code will be notified in the delegate method below. In this method, you should call MapView.setupMap.

```
/**
* Called when map view will begin loading. You must NOT call MapView.setupMap
before this method has been called.
* @param map the MapView
*/

@Override
public void willStartLoadingMapView(MapView map);
```

Map loading complete

In some scenarios, you may want to wait until the map view has completed loading before proceeding to the next step. You can wait for the didFinishLoadingMapView callback.

```
/**
* Called when map view has finished loading so you can do any additional setup
* @param map the MapView
*/

@Override
public void didFinishLoadingMapViews(MapView map);
```

Changes in map position

Anytime the map’s position changes, you can act accordingly.

NOTE: If your map view is overlaid on another provider's map (such as Google Maps), this is a good place to ensure the map positions stay in sync with each other.

```
/**
*
* @param newLocation the new map location
* @param newZoom the new map zoom
* @param newBearing the new map bearing
* @param newTilt the new map tilt
*/
@Override
public void didChangeCameraPosition(LatLng newLocation, float newZoom, double
newBearing, double newTilt);
```

Room selection/highlighting

When the user taps the map on a specific point or room, you can respond to those events appropriately. If you return true to canSelectUnit, the map will also highlight the selected shape using the color defined in your Map Style json for the layer-id of "floor_selected_unit_[FLOOR]". See “Customizing your map's look and feel” above for more information.

```
/**
* param mapUnit The map unit to check.
* @return Boolean indicating whether or not a map unit can be selected.
*/
Boolean canSelectUnit(MapUnit mapUnit);

/**
* Called when a map unit is selected on the map.
* @param mapUnit The map unit that was selected.
*/
void didSelectUnit(MapUnit mapUnit);
```

Adding your own annotations to the map view

You can programmatically add annotations to the map to suit your needs by creating a new PointAnnotation object and adding it to the map:

```
Bitmap icon = // custom icon;

PointAnnotation marker = new PointAnnotation();

marker.setFloorId(floor.getId()); //set the floor id to the VMMSBaseFloor
object's uid that this marker belongs on

marker.setTitle(wp.getId()); //give it a title, which you can use to reference
in additional callbacks

marker.setFloorNumber(floor.getFloorNumber());

marker.setPosition(target.getLocation()); //set the location you want the
annotation to appear on the map

//If your MapView has usingDefaultMapProvider == true

marker.setIcon(IconFactory.getInstance(this).fromBitmap(icon));

//If your MapView has usingDefaultMapProvider == false

marker.setHtml(“<div>Some html</div>”);

this.mapView.addMarker(marker);
```

NOTE: Custom annotations are implemented differently depending on whether your MapView has usingDefaultMapProvider set to true. If it is, then you can create a custom Bitmap to represent your map annotation. Otherwise, you will need to recreate your bitmap using HTML and CSS.

Responding to Errors that occur in the SDK

There are numerous instances where an error could occur within the VMSDK at any of the many steps above. You can be notified of the error by implementing any of the following callbacks:

VMD Parsing errors

If any errors are encountered while parsing your venue map data files, this method will be called within the SDK with more detailed information about the error:

```
/**
* Callback if the VMD fails to load
* @param e the exception that was raised during load
*/
@Override
public void didFailToLoadMapWithError(Exception e)
```

Map display errors

If any errors are encountered when your map is loaded and rendered on screen through the MapView object, this method will be called within the SDK with more detailed information about the error

```
/**
* Called when the map view fails to load
* @param mapView the MapView that failed to load
* @param ex the error that caused the failure
*/
@Override
public void didFailToLoadMapView(MapView mapView, Exception ex) {
```

Enable Wayfinding

You can add wayfinding capabilities to your map view to enable your application to provide turn-by-turn paths and directions:

```
//Create a new overlay that will display paths and markers for directions
this.walkingPathOverlay = new VectorWalkingPathOverlay(this.mapView);

//Set the delegate so this class can override certain functionality
this.walkingPathOverlay.setDelegate(this);

//NOTE: you have to wait until didFinishLoadingMap is called to be able to set
this!
this.walkingPathOverlay.setMap(this.vmd);
 .. and so on
```

Handle Wayfinding events

Implement the **com.aegir.vmms.wayfinding.WayfindingDelegate** interface to get notified of any callbacks from the SDK for wayfinding.

```
/**
* Callback after waypath has been determined
*
* @param path the Waypath that leads from the starting point to the ending
point
*/
@Override
public void didFinishFindingWaypath(Waypath path)
```

```
{
   this.vmd.createTurnByTurnDirectionsForWaypath(path, this.customMapInfo);
}

/**
* Called after turn by turn directions are done
*
* @param turnByTurnDirections the list of directions
*/
@Override
public void didFinishCreatingTurnByTurnDirections(List<MapDirectionStep>
turnByTurnDirections)
{
    //display path on the map now
    //tell your WalkingPathOverlay about the waypath  & directions
    this.walkingPathOverlay.setWaypathAndDirections(this.waypath, new
MapDirectionStepList(turnByTurnDirections));

    //Depending on your use-case, you might want to go ahead and
    //automatically change the map to display the floor corresponding
    //Figure out the first floor in the waypath
    WaypathSegment firstSegment = turnByTurnDirections.get(0).getSegment();
    MapBuildingFloor floor =
                 this.vmd.findFloorWithId(firstSegment.getFloorId());

    //update map view floors
    ...
}
```

Override auto-generated wayfinding directions and landmark names

You can use map information from a .json file to override the VMD's auto-generated wayfinding directions and landmark names. See this full example: **MapActivity.java.**

The data contained in your map info override file must be in JSON format, according to the following specs:

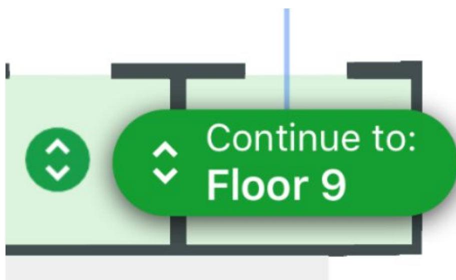
```
{
  "points": [
    {
      "id": "node_waypoint_b1_f1_517",
      "public-description": "the edge of the Basketball Court"
    },
    {
      "id": "<the ID of the waypoint>",
```

```
      "public-description": "<the description you want for this landmark/waypoint">
    }
  ],
  "paths": [
    {
      "pathID": "node_path_b1_f1_722",
      "p1": "node_waypoint_b1_f1_473",
      "p2": "node_waypoint_b1_f1_567",
      "description-d1": "along the sidewalk",
      "description-d2": "along the sidewalk the other direction"
    },
    {
      "pathID": "<the ID of the path>",
      "p1": "<the ID of one of the waypoints>",
      "p2": "<the ID of the other waypoint>",
      "description-d1": "<description for traversing from P1 to P2>, leave blank to auto-generate",
      "description-d2": "<description for traversing from P2 to P1>, leave blank to auto-generate"
    }
  ]
}
```

Responding to wayfinding events and callbacks

Changing floors for wayfinding

When you have a waypath that spans multiple floors, the default behavior for MapView is to draw a button on the map that looks like this:



You can provide your own image instead that matches your own branding. Additionally, when the user selects that button, you must implement that behavior as well by specifying what floor to change to, etc. See **MapActivity.java** for an example of how to appropriately respond to a floor change event.

```
/// Called to provide a custom image for the floor change annotation button
///
/// @param annotation the annotation
/// @return the custom image
@Override
public Bitmap imageForFloorChangeAnnotation(FloorChangePointAnnotation
annotation);

/**
* Called when a floor annotation is selected. You should use this to
* update the floor that is visible on your mapview
* @param annotation
*/
@Override
public void didSelectFloorChangeAnnotation(FloorChangePointAnnotation
annotation)
```

Responding to errors that occur in wayfinding

Wayfinding errors

Errors may occur during wayfinding, usually if no paths exist between your selected start & end destination. This method will be called within the SDK with more detailed information about the error:

```
/**
* Callback when wayfinding fails
* @param e the exception that was raised
*/
@Override
public void didFailToFindWaypathWithError(Exception e) {


/**
* Callback when turn-by-turn fails
* @param e the exception that was raised
*/
@Override
public void didFailToCreateTurnByTurnDirectionsWithError(Exception e)
```

Customizing wayfinding look and feel

All styling for wayfinding is now done through new styling properties added in v1.2 to the “wayfinding” section of the Vector map tile spec. For more information, see **Appendix: Vector map tile style spec** below.

Landmark customization

In the turn-by-turn directions provided by the SDK for wayfinding, there are usually points of interest, or landmarks, that are part of each step and refer to actual places on the map. You can add special icons to the map to further highlight your landmarks:

```
/**
 * Called to provide a custom image for a landmark annotation
 *
 * @param annotation the annotation
 * @return the custom image
 */

Bitmap imageForLandmarkAnnotation(LandmarkAnnotation annotation)s
```

NOTE: Landmark annotations are not currently supported if MapView has usingDefaultMapProvider set to false.

More Information

For assistance with the Aegir VMSDK, related questions, or information about other Aegir products and services, visit https://support.aegirmaps.com/, contact Aegir Support at support@aegirmaps.com or call us at (901) 591-1631 between 9:00 am and 5:00 pm CST, Monday through Friday.

Appendix: Vector Map Tile Style Spec

Supported spec versions: 1.0, 1.1*

As new styleable features are added, we will attempt to maintain backwards compatibility with older versions of this specification, however newer styling features may not be available in older versions.

*Some style properties are not supported when your MapView has usingDefaultMapProvider set to false.

Style spec properties

Property-name	Required	Supported layer-type	Spec version	Description
id	yes		1.0	Identifier for this style definition. For venues with multiple styles, this identifier should be unique
name	no		1.0	Common name used to describe this style
version	no		1.1*	Spec format version. Certain versions of the SDK may only support certain versions of this spec format. Default: 1.0
styles	yes		1.0	Container for list of style layer customizations
styles[].layer-id	yes	All	1.0	The ID of the style layer. See Possible style layers section below for acceptable values.
styles[].hidden	no	All	1.0	Specify as true to hide this layer. Default: false.
styles[].fill-color	no	Polygon	1.0	A HEX color to fill the polygon with. Default: NULL.
styles[].fill-pattern	no	Polygon	1.0	This is the name of an image from the style's sprite sheet to pattern fill the polygon with. Default: NULL.

Property-name	Required	Supported layer-type	Spec version	Description
styles[].outline-color	no	Polygon	1.1*	This is a HEX color to draw an outline around the polygon with. Default: NULL
styles[].line-color	no	Line	1.0	A HEX color to draw the line with. Default: NULL.
styles[].icon-name	no	Icon	1.0	This is the name of an image from the style's sprite sheet. Default: NULL.
styles[].font-name	no	Label	1.0	This is the name of a font to use for the labels. Default: NULL.
styles[].font-size	no	Label	1.0	The point size of the font. Default: NULL.
styles[].font-color	no	Label	1.0	A HEX color of the font: Default: NULL.
styles[].font-stroke-color	no	Label	1.0	A HEX color for the font outline. Default: NULL.
styles[].font-stroke-width	no	Label	1.0	This is the width of an outline for the font. Default: NULL.
styles[].max-text-width	no	Label	1.0	Controls automatic text wrapping within a label. Default: NULL.
wayfinding	no		1.0	Container for list of wayfinding style customizations.
wayfinding.path-stroke-width	no		Deprecated 1.1	A decimal value indicating how thick the stroke is for the default wayfinding path. Default: NULL.
wayfinding.path-stroke-min-width	no		1.1*	This is a decimal value indicating how thick the stroke is for the default wayfinding path at the map's minimum zoom level. Default: 2
wayfinding.path-stroke-max-width	no		1.1*	This is a decimal value indicating how thick the stroke is for the default wayfinding path at the map's maximum zoom level. Default: 40

wayfinding.path-stroke-color	no		1.0	A HEX color to draw the wayfinding path. Default: NULL.
wayfinding.path-stroke-alpha	no		1.0	A decimal value from 0 to 1 to indicate how transparent the default wayfinding path is. Default: NULL.
wayfinding.path-arrow-fill-color	no		1.1*	This is a HEX color to draw the arrows on the wayfinding path. Default: #4688F1
wayfinding.path-arrow-stroke-color	no		1.1*	This is a HEX color of the outline around the arrows drawn on the wayfinding path.Default: #FFFFFF
wayfinding.path-arrow-size	no		1.1*	This is a decimal value indicating how large the arrows on the wayfinding path show up. The value should be in METERS. Default: 1.5
wayfinding.highlighted-path-stroke-width	no		Deprecated 1.1	A decimal value indicating how thick the stroke is for the highlighted section of the wayfinding path. Default: NULL.
wayfinding.highlighted-path-stroke-min-width	no		1.1*	This is a decimal value indicating how thick the stroke is for the highlighted section of the wayfinding path at the map's minimum zoom level. Default: 2
wayfinding.highlighted-path-stroke-max-width	no		1.1*	This is a decimal value indicating how thick the stroke is for the highlighted section of the wayfinding path at the map's maximum zoom level. Default: 40
wayfinding.highlighted-path-stroke-color	no		1.0	A HEX color to draw a highlighted section of the wayfinding path. Default: NULL.
wayfinding.highlighted-path-stroke-alpha	no		1.0	A decimal value from 0 to 1 to indicate how transparent the highlighted section of the wayfinding path is. Default: NULL.
wayfinding.highlighted-path-arrow-fill-color	no		1.1*	This is a HEX color of the arrows drawn in a highlighted section of the wayfinding path. Default: #4688F1

wayfinding.highlighted-path-arrow-stroke-color	no		1.1*	This is a HEX color of the outline around the arrows drawn in a highlighted section of the wayfinding path.Default: #FFFFFF
wayfinding.highlighted-path-arrow-size	no		1.1*	This is a decimal value indicating how large the arrows in a highlighted section of the wayfinding path. The value should be in METERS. Default: 1.5

Possible Style Layer ID Patterns

This is a list of possible layer-IDs that can be styles per the style spec above. Some layers apply only to raster or vector, while others apply to both. This is indicated in the 'tile-type' column below. This list is ordered by the zIndex they would appear in the map: layers with a higher Order will appear on top of those with lower values.

Supported wildcards:

1. FLOOR - the ID of the floor layer from the VMD (e.g. floor_b1_1)
2. BUILDING - the ID of the building layer from the VMD (e.g. building_1)

When wildcards are used, the specific style will be applied to all layers that match. For example, floor_elevators_[FLOOR] will be used for the floor_elevators_* layer on ALL floors in ALL buildings.

If you want to confine a unique style to a layer on a single floor, then don't use the wildcard. For example floor_elevators_floor_b1_2 would apply to the floor_elevators layer ONLY on floor 2 in building 1.

Order	Layer-ID	Tile-type	Layer-type	Description
1	background	All	Polygon	Background color of the entire map that is visible when the base map (Google Maps or Apple Maps) is hidden.
2	venue	vector	Polygon	“Venue_outdoors” polygon.
3	outdoors	All	n/a	Raster map tiles that are part of the venue outdoor floor.
4	building_outlines_[BUILDING]	vector	Polygon	Building-outlines polygon for the given [BUILDING].
5	floor_outlines_[FLOOR]	vector	Polygon	Floor-outlines polygon for the given [FLOOR].
6	floor_elevators_[FLOOR]	vector	Polygon	Elevator polygons for the given [FLOOR].
7	floor_stairwells_[FLOOR]	vector	Polygon	Stairwell polygons for the given [FLOOR].
8	floor_restrooms_[FLOOR]	vector	Polygon	“Restroommen” and “restroomwomen” polygons for the given [FLOOR].

Order	Layer-ID	Tile-type	Layer-type	Description
9	floor_walkways_[FLOOR]	vector	Polygon	Walkway polygons for the given [FLOOR].
10	floor_fixtures_[FLOOR]	vector	Polygon	Floor fixture polygons for the given [FLOOR].
11	floor_non_public_units_[FLOOR]	vector	Polygon	Non-public unit polygons for the given [FLOOR].
12	floor_open_to_below_units_[FLOOR]	vector	Polygon	“Open to below” unit polygons (such as open atrium spaces) for the given [FLOOR].
13	floor_other_rooms_[FLOOR]	vector	Polygon	“Other room” polygons for the given [FLOOR].
14	floor_rooms_[FLOOR]	vector	Polygon	Room polygons for the given [FLOOR].
15	floor_water_[FLOOR]	vector	Polygon	Fixtures where category=Water for the given [FLOOR].
16	floor_openings_[FLOOR]	vector	Line	Floor openings for the given [FLOOR].
17	floor_amenities_[FLOOR]	vector	**	Floor amenities for the given [FLOOR].
18	floor_selected_unit_[FLOOR]	vector	Polygon	This is for the style of the actively selected polygon used during wayfinding & room selection for the given [FLOOR].
19	floor_shadows_[FLOOR]	vector	n/a	Raster map tiles that are overlaid on top of existing vector data for the given [FLOOR]
20	[FLOOR]	Raster	n/a	Raster map tiles for the given floor [FLOOR].
21	building_outlines_[BUILDING]	Raster	n/a	Raster map tiles for the given building [BUILDING].
22	floor_labels_[FLOOR]	All	Label	Labels for the given [FLOOR].
23	floor_icons_[FLOOR]	All	Icon	Icons for the given [FLOOR].
24	building_labels_[BUILDING]	All	Label	These are the labels for a given [BUILDING] that are displayed when no active floors in that building are shown.