**IVI Location Based Service API Design Best Practices**

Contents

[**Map API** 3](#_Toc461994863)

[**POI API** 8](#_Toc461994871)

[**Driving API** 10](#_Toc461994876)

[**RoadInfoSearch API** 15](#_Toc461994884)

[**Geocoder API** 19](#_Toc461994891)

**Map API**

Map API packages a set of function, like map display, user interaction, etc.

**Interface**

**Interface 'Map'**

[Constructor(Node mapDiv, MapOptions opts)]

interface Map {

Promise<any> setProperty(DOMString propertyName, any newValue);

Promise<any> getProperty(DOMString propertyName);

Bounds getBounds();

void setBounds(Bounds bound);

long getResolution(LngLat point);

void zoomIn();

void zoomOut();

void panTo(LngLat position);

void panBy(long x, long y);

LngLat pixelToContainer(Pixel pixel, long level);

Pixel containerToPixel(LngLat lnglat, long level);

};

Map is the main interface to interact with a map. A Map instance can be returned by using the Map constructor that accepts a 'Node' and a MapOptions as input.

* setProperty(): set a new value for a property defined in the map options.
* getProperty(): get the value of a property defined in the map options.
* getBounds(): get the view bounds of current map. See Bounds Interface for more details about the possible values.
* setBounds(): set the view bounds of current map. See Bounds Interface for more details about the possible values.
* getResolution(): Get resolution of given point, unit is meter per pixel. If point is unset, then return the resolution of map.
* zoomIn(): Zoom in map.
* zoomOut(): Zoom out map.
* panTo(): Pan the map center point to given position as input.
* panBy(): Pan the map along x direction and y direction, by pixel. x is positive rightward, y is positive downward.
* pixelToContainer(): Convert the map pixel coordinates to screen coordinates, according to map zoom level.
* containerToPixel(): Convert the screen coordinates to map pixel coordinates, according to map zoom level.

**Interface 'MapOptions'**

dictionary MapOptions {

LngLat center;

long zoom;

Boolean isHotspot;

Boolean rotateEnable;

Boolean dragEnable;

Boolean zoomEnable;

Boolean jogEnable;

};

A 'MapOptions' is a dictionary that is passed as input to the 'Map' constructor. It may be extended in the future with additional option properties. The current supported option properties are:

* center: Coornidates of map center.See LngLat Interface for more details about the possible values.
* zoom: It defines the zoom level of map.
* isHotspot: It defines whether to open map hotspot function, default is false.
* rotateEnable: It defines whether to open map rotating function, default is false.
* dragEnable: It defines whether to open map dragging or panning function, default is true.
* zoomEnable: It defines whether to open map zooming function, default is true.
* jogEnable: It defines whether to use jog effect, default is true.

**Interface 'LngLat'**

[Constructor(float lng, float lat)]

interface LngLat {

float lng;

float lat;

LngLat offset(long w, long s);

long distance(LngLat lnglat);

float getLng();

float getLat();

};

LngLat is used to define a point on map. A LngLat instance can be returned by using the LngLat constructor that accepts a longitude and a latitude as input.

* lng: longitude value
* lat: latitude value
* offset(): Get new coordinate value by moving w meters in longitude and s meters in latitude. w is positive rightward, s is positive upward.
* distance(): Get the distance between current lnglat and this passed-in lnglat. Unit is meter.
* getLng(): Get current longitude value.
* getLat(): Get current latitude value.

**Interface 'Bounds'**

[Constructor(LngLat southWest, LngLat northEast)]

interface Bounds {

Boolean contains(LngLat point);

LngLat getCenter();

LngLat getSouthWest();

LngLat getNorthEast();

}

Bounds is used to define a rectangle bound on map. A Bounds instance can be returned by using the Bounds constructor that accepts a southwest point coordinates value and a northeast point coordinates as input.

* contains(): Get that whether the passed-in point is within the range of bound.
* getCenter(): Get center coordinates of bound.
* getSouthWest(): Get southwest point coordinates of bound.
* getNorthEast(): Get northeast point coordinates of bound.

**Interface 'Pixel'**

[Constructor(long x, long y)]

interface Pixel {

long getX();

long getY();

}

Pixel is used to define a pixel object on map. A Pixel instance can be returned by using the Pixel constructor that accepts a x direction value and y direction value as input.

* getX(): Get the x direction value of Pixel object.
* getY(): Get the y direction value of Pixel object.

**Example Create Map instance**

var map = new AMap.Map('mapContainer',{

rotateEnable: true,

//Set map center point

center: new LngLat(116.397428, 39.90923),

//Set zoom level

zoom: 12

});

**POI API**

**POI API example**

**Interface 'POI'**

[Constructor(PlaceSearchOptions opts)]

interface PlaceSearch {

Promise<any> setProperty(DOMString propertyName, any newValue);

Promise<any> getProperty(DOMString propertyName);

void search(DOMString keyword, SearchCallback whenSearched);

void searchNearBy(DOMString keyword, LngLat center, long radius, SearchCallback whenSearched);

void searchInBounds(DOMString keyword, Bounds bounds, SearchCallback whenSearched);

};

callback SearchCallback = void(DOMString status, (sequence<Poi> or DOMString) result);

PlaceSearch is the main interface to deal with POI search. A PlaceSearch instance can be returned by using the PlaceSearch constructor that accepts a PlaceSearchOptions as input.

* setProperty(): set a new value for a property defined in the PlaceSearch options.
* getProperty(): get the value of a property defined in the PlaceSearch options.
* search(): Search the POI according to keyword. Result will be notified through the callback. See SearchResult Interface for more details about the possible values.
* searchNearBy(): Search the POI according to circular region (center point coordinates, radius) and keyword. Result will be notified through the callback. See SearchResult Interface for more details about the possible values.
* searchInBounds(): Search the POI according to rectangular region and keyword. Result will be notified through the callback. See SearchResult Interface for more details about the possible values.

**POI Data format example**

**Interface 'Poi'**

interface Poi {

readonly attribute DOMString id;

readonly attribute DOMString name;

readonly attribute DOMString type;

readonly attribute LngLat location;

readonly attribute DOMString address;

readonly attribute long distance;

readonly attribute DOMString tel;

readonly attribute DOMString website;

readonly attribute DOMString pcode;

readonly attribute DOMString citycode;

readonly attribute DOMString adcode;

readonly attribute DOMString postcode;

readonly attribute DOMString pname;

readonly attribute DOMString cityname;

readonly attribute DOMString adname;

readonly attribute DOMString email;

readonly attribute LngLat entr\_location;

readonly attribute LngLat exit\_location;

};

**Driving API**

Driving API packages a set of function, to provide driving route planning service.

**Interface**

**Interface 'Driving'**

[Constructor(DrivingOptions opts)]

interface Driving {

Promise<any> setProperty(DOMString propertyName, any newValue);

Promise<any> getProperty(DOMString propertyName);

void search(LngLat origin, LngLat destination, optional sequence<LngLat> viaPoints, SearchCallback whenSearched);

void search(DOMString origin, DOMString destination, optional sequence<DOMString> viaPoints, SearchCallback whenSearched);

void setAvoidRoad(DOMString road);

DOMString getAvoidRoad();

void clearAvoidRoad();

void clear();

};

callback SearchCallback = void(DOMString status, (DrivingResult or DOMString) result);

Driving is the main interface to deal with driving route planning. A Driving instance can be returned by using the Driving constructor that accepts a DrivingOptions as input.

* setProperty(): set a new value for a property defined in the Driving options.
* getProperty(): get the value of a property defined in the Driving options.
* search(): Search the driving planning result according to orgin coordinates or name, destination coordinates or name as well as optional via points coordinates or names. Result will be notified through the callback. If status is ‘error’, then error info is returned; if status is ‘complete’, then driving route result is returned. See DrivingResult Interface for more details about the possible values.
* setAvoidRoad(): Set the road which will be avoided during route planning.
* getAvoidRoad(): Get the road name which will be avoided during route planning.
* clearAvoidRoad(): Clear the road which will be avoided during route planning.
* clear(): Clear the route planning result.

**Interface 'DrivingOptions'**

dictionary DrivingOptions {

DrivingPolicy policy;

DOMString extensions;

};

A 'DrivingOptions' is a dictionary that is passed as input to the 'Driving' constructor. It may be extended in the future with additional option properties. The current supported option properties are:

* policy: Policy of driving route planning. See DrivingPolicy Interface for more details about the possible values.
* extensions: Choose either 'base' or 'all'. ‘base’: default, return basic driving planning info; ‘all’: return basic and optional detailed driving planning info.

**Interface 'DrivingPolicy'**

enum DrivingPolicy {

"LEAST\_TIME",

"LEAST\_FEE",

"LEAST\_DISTANCE",

"REAL\_TRAFFIC"

};

The DrivingPolicy is a enumaration about possible values for the Driving.policy attribute. The four values are currently supported:

* LEAST\_TIME: Least time consuming
* LEAST\_FEE: Least cost
* LEAST\_DISTANCE: Least distance
* REAL\_TRAFFIC: Driving route planning considering the real traffic condition

**Interface 'DrivingResult'**

interface DrivingResult {

readonly attribute DOMString info;

readonly attribute LngLat orgin;

readonly attribute LngLat destination;

readonly attribute Poi originPoi;

readonly attribute Poi destinationPoi;

readonly attribute sequence<DriveRoute> routes;

};

The DrivingResult is driving route planning result returned.

* info: Description of success.
* orgin: Origin coordinates of planned driving route.
* destination: Destination coordinates of planned driving route.
* originPoi: Origin POI details of planned driving route.
* destinationPoi: Destination POI details of planned driving route.
* routes: List of routes. See DriveRoute Interface for more details about the possible values.

**Interface 'DriveRoute'**

interface DriveRoute {

readonly attribute long distance;

readonly attribute long time;

readonly attribute DrivingPolicy policy;

readonly attribute sequence<DriveStep> steps;

};

* distance: Distance from origin to destination. Unit is meter.
* time: Estimated time consuming. Unit is second.
* policy: Policy of driving route planning.
* steps: List of each driving route step. See DriveStep Interface for more details about the possible values.

**Interface 'DriveStep'**

interface DriveStep {

readonly attribute LngLat start\_location;

readonly attribute LngLat end\_location;

readonly attribute DOMString instruction;

readonly attribute DOMString action;

readonly attribute DOMString orientation;

readonly attribute DOMString road;

readonly attribute DOMString distance;

readonly attribute long time;

readonly attribute sequence<DOMString> cities;

};

* start\_location: Start location coordinates of this driving route step.
* end\_location: End location coordinates of this driving route step.
* instruction: Instruction of how to drive through this route step.
* action: What to do next after arriving the end location of this route step.
* orientation: Driving orientation.
* road: Road name.
* distance: Distance of this route step.
* time: Estimated time consuming of this route step. Unit is second.
* cities: Name of via cities.

# RoadInfoSearch API

RoadInfoSearch API packages a set of function, to provide road info search service.

# Interface

## Interface 'RoadInfoSearch'

[Constructor(DOMString city)]

interface RoadInfoSearch {

attribute DOMString city;

void roadInfoSearchByRoadId(DOMString id, SearchCallback whenSearched);

void roadInfoSearchByRoadName(DOMString keyword, SearchCallback whenSearched);

void crossInfoSearchByCrossId(DOMString id, SearchCallback whenSearched);

void crossInfoSearchByCrossName(DOMString keyword, SearchCallback whenSearched);

void setCity(DOMString city);

};

callback SearchCallback = void(DOMString status, (RoadInfoSearchResult or CrossInfoSearchResult or DOMString) result);

RoadInfoSearch is the main interface to deal with road info search. A RoadInfoSearch instance can be returned by using the RoadInfoSearch constructor that accepts a DOMString as input.

* city: City name or city code. The city within which to search the road info. If more than one, separate by ‘|’.
* roadInfoSearchByRoadId(): Search the road info result according to road id. Result will be notified through the callback. If status is ‘error’, then error info is returned; if status is ‘complete’, then road info result is returned. See RoadInfoSearchResult Interface for more details about the possible values.
* roadInfoSearchByRoadName(): Search the road info result according to road name. Result will be notified through the callback. If status is ‘error’, then error info is returned; if status is ‘complete’, then road info result is returned. See RoadInfoSearchResult Interface for more details about the possible values.
* crossInfoSearchByCrossId(): Search the crossroad info result according to crossroad id. Result will be notified through the callback. If status is ‘error’, then error info is returned; if status is ‘complete’, then road info result is returned. See CrossInfoSearchResult Interface for more details about the possible values.
* crossInfoSearchByCrossName(): Search the crossroad info result according to crossroad name. Result will be notified through the callback. If status is ‘error’, then error info is returned; if status is ‘complete’, then road info result is returned. See CrossInfoSearchResult Interface for more details about the possible values.
* setCity(): Set the city within which to do searching.

## Interface 'RoadInfoSearchResult'

interface RoadInfoSearchResult {

readonly attribute DOMString info;

readonly attribute sequence<RoadInfo> roadInfoList;

readonly attribute sequence<DOMString> keywordList;

readonly attribute sequence<DOMString> cityList;

};

The RoadInfoSearchResult is road info result returned.

* info: Description of success.
* roadInfoList: List of road info returned. See RoadInfo Interface for more details about the possible values.
* keywordList: If no match found, return the list of recommended keywords.
* cityList: If no match found in the chosen city, return the list of recommended cities.

## Interface 'RoadInfo'

interface RoadInfo {

readonly attribute DOMString id;

readonly attribute DOMString name;

readonly attribute LngLat center;

readonly attribute DOMString citycode;

readonly attribute sequence<LngLat> path;

};

* id: Road id.
* name: Road name.
* center: Coordinates of road center.
* citycode: City code.
* path: List of point coordinates through the road.

## Interface 'CrossInfoSearchResult'

interface CrossInfoSearchResult {

readonly attribute DOMString info;

readonly attribute sequence<CrossInfo> roadInfoList;

readonly attribute sequence<DOMString> keywordList;

readonly attribute sequence<DOMString> cityList;

};

The CrossInfoSearchResult is crossroad info result returned.

* info: Description of success.
* roadInfoList: List of crossroad info returned. See CrossInfo Interface for more details about the possible values.
* keywordList: If no match found, return the list of recommended keywords.
* cityList: If no match found in the chosen city, return the list of recommended cities.

## Interface 'CrossInfo'

interface CrossInfo {

readonly attribute DOMString id;

readonly attribute LngLat location;

readonly attribute DOMString firstRoadId;

readonly attribute DOMString firstRoadName;

readonly attribute DOMString secondRoadId;

readonly attribute DOMString secondRoadName;

readonly attribute DOMString citycode;

};

* id: Crossroad id.
* location: Crossroad location coordinates.
* firstRoadId: Road id of one of the roads.
* firstRoadName: Road name of one of the roads.
* secondRoadId: Road id of the other road.
* secondRoadName: Road name of the other road.
* citycode: City code.

# Geocoder API

Geocoder API packages a set of function, to provide Forward Geocoding and Reverse Geocoding service.

# Interface

## Interface 'Geocoder'

[Constructor(GeocoderOptions opts)]

interface Geocoder {

getLocation(DOMString address, SearchCallback whenSearched);

getAddress((LngLat or sequence<LngLat>) location, SearchCallback whenSearched);

void setCity(DOMString city);

};

callback SearchCallback = void(DOMString status, (GeocodeResult or ReGeocodeResult or DOMString) result);

Geocoder is the main interface to deal with Forward Geocoding and Reverse Geocoding service. A Geocoder instance can be returned by using the Geocoder constructor that accepts a GeocoderOptions as input.

* getLocation(): Forward geocoding. Obtain the geographic coordinates according to address description. Result will be notified through the callback. If status is ‘error’, then error info is returned; if status is ‘complete’, then location info result is returned. See GeocodeResult Interface for more details about the possible values.
* getAddress(): Reverse geocoding. Obtain the address description according to geographic coordinates. Result will be notified through the callback. If status is ‘error’, then error info is returned; if status is ‘complete’, then road info result is returned. See ReGeocodeResult Interface for more details about the possible values.
* setCity(): Set the city within which to do searching.

## Interface 'GeocoderOptions'

dictionary GeocoderOptions {

attribute DOMString city;

attribute long radius;

DOMString extensions;

};

A 'GeocoderOptions' is a dictionary that is passed as input to the 'Geocoder' constructor. It may be extended in the future with additional option properties. The current supported option properties are:

* city: City of Geocoder. Format can be as city name or city code or administrative code.
* radius: Used in reverse geocoding. Set the range of searching area.
* extensions: Used in reverse geocoding. Choose either 'base' or 'all'. ‘base’: default, return basic address info; ‘all’: return basic and optional detailed address info.

## Interface 'GeocodeResult'

interface GeocodeResult {

readonly attribute DOMString info;

readonly attribute sequence<LngLat> location;

};

The GeocodeResult is getLocation result returned.

* info: Description of success.
* geocodes: location list.

## Interface 'ReGeocodeResult'

interface ReGeocodeResult {

readonly attribute DOMString info;

readonly attribute DOMString address;

};

The ReGeocodeResult is getAddress result returned.

* info: Description of success.
* regeocode: address description result.