Tracking Position and Heading

Objective

In this chapter, you'll learn how to use Titanium's Geolocation API to retrieve GPS positioning and heading information from mobile devices. You'll learn iOS, Android, and Windows Phone specifics that will help you best balance accuracy with battery consumption. And you'll learn how to manage Geolocation listeners with respect to your application's lifecycle.

Contents

The position and heading APIs are part of the `Ti.Geolocation` module, which contains all the functions, properties, and events necessary to handle location information. That namespace is further divided into the `Ti.Geolocation.Android` namespaces, which provide Android-specific features. In the following sections, you'll learn how to use this API to perform the following activities:

- Detect if location services are available
- Obtain the current GPS position
- Continually monitor the GPS position
- Use the device's compass
- Configure location service properties
- Forward and reverse geocoding

You'll also learn best practices and caveats to consider when using location services in your apps. But first, let's dive into some platform specific considerations when using these services.

Development considerations

iOS development considerations

iOS users are prompted to grant or deny permission when your application attempts to use geolocation information.

Starting with iOS 8, to use location services, add either the `NSLocationWhenInUseUsageDescription` or `NSLocationAlwaysUsageDescription` key to the iOS plist section of the project's `tiapp.xml` file. To localize the message, see Internationalization: Localize Property List Keys.

Starting with iOS 11, to request Always permission you must replace the `NSLocationAlwaysUsageDescription` with `NSLocationAlwaysAndWhenInUseUsageDescription`. This is because your users now have the ability to still choose "When in use" when the app is requesting the "Always" permission. In the flow of your app you need to take this into consideration.

```
<ti:app>
    <ios>
        <plist>
            <dict>
                <key>NSLocationAlwaysUsageDescription</key>
                <string>
                    Specify the reason for accessing the user's location information.
                    This appears in the alert dialog when asking the user for
permission to
                    access their location.
                </string>
            </dict>
        </plist>
    </ios>
</ti:app>
```

Android development considerations

In general, testing geolocation code should be done on a device so that you accurately and realistically test your app in an environment close to a real-world usage scenario. However, Emulators created using Android Studio 3 support GPS out the box too.

Starting at TiSDK 7.1.0.GA there is support for FusedLocationProvider. To enable this, the only thing you need to do is include the Ti.PlayServices module in your app. This will enable battery efficient geolocation API's in your app.

Windows development considerations

> Support for Windows 8.1 and Windows Phone SDKs has been deprecated as of SDK 6.3.0.GA and has be removed in SDK 7.0.0.GA.

In order to enable location service for Windows Phone, you need to provide appropriate `location` Capability in your `tiapp.xml`. Windows Phone users are prompted to grant or deny permission when your application attempt to use geolocation information. In any cases Windows Phone user should enable location service on their device preliminarily (`Settings -> location` on Windows Phone, `Settings -> Privacy -> Location` on Windows 10 Mobile).

```
<ti:app>
  ...
  <windows>
    ...
    <manifest>
      <Capabilities>
        <DeviceCapability Name="location" />
      </Capabilities>
    </manifest>
    ...
  </windows>
  ...
</ti:app>
```

For more information about geolocation configuration in `tiapp.xml`, see Windows-specific section in tiapp.xml and timodule.xml Reference.

Using location services in your app

Using location services generally follows these three stages:

1. Requesting Location Permission

2. Determine if location services are enabled and available.
3. Configure the accuracy and listener mechanisms to use.
4. Grab a one-time location or enable a location-listener to continually monitor a user's location.
5. With a location-listener, actively manage the listener in coordination with the app's lifecycle.

Using location services can have a significant impact on a device's battery life, so it's important to use them in the most efficient manner possible. Power consumption is strongly influenced by the accuracy and frequency of location updates required by your application. The higher the accuracy you request, and the more frequently you request location updates, the more battery power that will be consumed.

Requesting location permission

When you want to use Geolocation in your app you'll need to ask for permission of your user. On Android this needs to be requested starting version 6.0. Before Android 6.0 all was needed was the geolocation permission in the manifest.

It is pretty straightforward to request permission. For iOS you need to configure your plist correctly as described at the iOS Development Considerations section. Before asking permission it is advised to check if the permission is already given. In the sample below you can see how to do all this, we're going to be requesting for permisson while the app is in use.

```
var hasLocationPermission =
Ti.Geolocation.hasLocationPermissions(Ti.Geolocation.AUTHORIZATION_WHEN_IN_USE);
if (!hasLocationPermission) {
 Ti.Geolocation.requestLocationPermissions(Ti.Geolocation.AUTHORIZATION_WHEN_IN_USE,
function(e) {
  if (e.success) {
   // permission granted
  } else {
   // permission refused
  }
 }
}
```

Note: Only on iOS is the attribute `Ti.Geolocation.AUTHORIZATION_WHEN_IN_USE` is required. The property is ignored on other platforms.

On iOS you can find out why the permission has been refused. A very thourough example for requesting permissions was created in the 5.1.0 sample app, which is available on GitHub https://github.com/appcelerator-developer-relations/appc-sample-ti510/blob/master/app/controllers/permissions.js

Detect if location services are available

To determine whether or not location services will be available to you on the current mobile device, you simply need to check the boolean property `Ti.Geolocation.locationServicesEnabled`. Keep in mind, though, that on Android 2.2 and above, a low-precision "passive" location provider is enabled at all times, even when the user disables both the GPS and Network location providers. Therefore, this method always returns `true` on such devices. With this in mind, the base skeleton of a locations based app might look something like this.

```
if (Ti.Geolocation.locationServicesEnabled) {
    // perform other operations with Ti.Geolocation
} else {
    alert('Please enable location services');
}
```

Configure the accuracy and frequency

The location services systems of the underlying platforms are very different, so there are significant implementation differences between the platforms. The basic methods of requesting location information and receiving location updates are essentially the same on all platforms. However, the method of configuring the accuracy and frequency of location updates is different for each platform.

iOS geo configuration

In iOS, the accuracy (and power consumption) of location services is primarily determined by the `Ti.Geolocation.accuracy` property setting. You can set this property to one of the following values:

- ACCURACY_BEST (highest accuracy and power consumption)

- ACCURACY_NEAREST_TEN_METERS
- ACCURACY_HUNDRED_METERS
- ACCURACY_KILOMETER
- ACCURACY_THREE_KILOMETERS (lowest accuracy and power consumption).

(Note that the constants ACCURACY_HIGH and ACCURACY_LOW are Android-specific and may not be used with iOS.)

Based on the accuracy you choose, iOS uses its own logic to select location providers and filter location updates to provide location updates that meet your accuracy requirements. You can further limit power consumption on iOS by setting the `Ti.Geolocation.distanceFilter` property to eliminate position updates when the user is not moving. That property accepts a distance in meters; when the user has moved approximately that distance, your app will receive location update events.

- **accuracy** - The target accuracy of all location data received. The following `Ti.Geolocation`constants represent the valid values for this property:
    - `ACCURACY_BEST` - Location data will be of the highest possible accuracy of which the device is capable
    - `ACCURACY_HUNDRED_METERS` - Location data will be accurate within 100 meters
    - `ACCURACY_KILOMETER` - Location data will be accurate within 1 kilometer
    - `ACCURACY_NEAREST_TEN_METERS` - Location data will be accurate within 10 meters
    - `ACCURACY_THREE_KILOMETERS_` - Location data will be accurate within 3 kilometers
- **distanceFilter** - The minimum change of position (in meters) before a `location` event is fired. The default is 0, meaning that location events are continuously generated.
- **headingFilter** - The minium change of heading (in degrees) before a `heading` event is fired. The default is 0, meaning that heading events are continuously generated.
- **preferredProvider** - Allows you to specify the preferred method for receiving a location. The following `Ti.Geolocation`constants represent your possible choices:
    - `PROVIDER_NETWORK` - Give the network based location provider preference
    - `PROVIDER_GPS` - Give the GPS location preference

Using the event-driven location example at the beginning of this chapter, let's modify it to use some of the above properties.

Geolocation configuration on iOS

```
if (Ti.Geolocation.locationServicesEnabled) {
    Ti.Geolocation.purpose = 'Get Current Location';
    Ti.Geolocation.accuracy = Ti.Geolocation.ACCURACY_BEST;
    Ti.Geolocation.distanceFilter = 10;
    Ti.Geolocation.preferredProvider = Ti.Geolocation.PROVIDER_GPS;

    Ti.Geolocation.addEventListener('location', function(e) {
        if (e.error) {
            alert('Error: ' + e.error);
        } else {
            Ti.API.info(e.coords);
        }
    });
} else {
    alert('Please enable location services');
}
```

Android geo configuration

Since Android offers a much richer geolocation model, with multiple location providers, distance filters, update frequencies, and so forth we offer a method of using manual and simple mode for geolocation.

- **Manual mode** gives developers low-level control of location updates, including enabling individual location providers and filtering updates, for the best combination of accuracy and battery life. Manual mode is used when the `Titanium.Geolocation.Android.manualMode` flag is set to `true`. In manual mode, the accuracy property is not used, and all configuration is done through the `Titanium.Geolocation.Android` module.
- **Simple mode** provides a compromise mode that provides adequate support for undemanding location applications without requiring developers to write a lot of Android-specific code. Setting `Ti.Geolocation.accuracy` to either ACCURACY_HIGH or ACCURACY_LOW enables simple mode. In this mode the platform handles enabling and disabling location providers and filtering location updates.

```
// demonstrates manual mode:
var providerGps = Ti.Geolocation.Android.createLocationProvider({
    name: Ti.Geolocation.PROVIDER_GPS,
    minUpdateDistance: 0.0,
    minUpdateTime: 0
});
Ti.Geolocation.Android.addLocationProvider(providerGps);
Ti.Geolocation.Android.manualMode = true;
var locationCallback = function(e) {
    if (!e.success || e.error) {
        Ti.API.info('error:' + JSON.stringify(e.error));
    } else {
  Ti.API.info('coords: ' + JSON.stringify(e.coords));
 }
};
Titanium.Geolocation.addEventListener('location', locationCallback);
```

See the https://docs.appcelerator.com/platform/latest/#!/api/Titanium.Geolocation.Android for further Android-specific information.

Obtain the current GPS position

With your app configured to use the appropriate level of platform-specific geolocation configuration, you're ready to work with location data. Many apps only infrequently need to use location services. Whether it's at app startup, on a button click, or at a timed interval, developers have a multitude of opportunities to actively query for location information.

Let's take a look at a very basic example. After asserting that location services are enabled and permissions are requested, the `Ti.Geolocation.getCurrentPosition()` function is used to query for location information. This function takes a single parameter; a callback function whose event object contains the requested location in its `coords` property. This is an asynchronous call as the GPS functionality may take a moment to work, especially if this is the first time your app is accessing location. Also worth noting is that the location services might return a cached location (depending on the platform and the configuration choices you have made).

```
if (Ti.Geolocation.locationServicesEnabled) {
    Titanium.Geolocation.getCurrentPosition(function(e) {
        if (e.error) {
            Ti.API.error('Error: ' + e.error);
        } else {
            Ti.API.info(e.coords);
        }
    });
} else {
    alert('Please enable location services');
}
```

The output for a successful execution of the above app would look something like this:

```
{
    "accuracy": 100,
    "altitude": 0,
    "altitudeAccuracy": null,
    "heading": 0,
    "latitude": 40.493781233333333,
    "longitude": -80.056671
    "speed": 0,
    "timestamp": 1318426498331
}
```

Continually monitor the GPS position

Often you will want to know where a mobile device is at all times. The most common example of this is navigation for driving directions. To have the same constant awareness of a device's position in Titanium, you simply need to monitor the `location` event with the `Ti.Geolocation` module.

Here's a simple case showing how location data can be handled via event listener. You'll notice that the data is handled in a nearly identical manner to the `Ti.Geolocation.getCurrentPosition()` example.

```
if (Ti.Geolocation.locationServicesEnabled) {
    Ti.Geolocation.addEventListener('location', function(e) {
        if (e.error) {
            alert('Error: ' + e.error);
        } else {
            Ti.API.info(e.coords);
        }
    });
} else {
    alert('Please enable location services');
}
```

As with the `Ti.Geolocation.getCurrentPosition()` example, the location data is returned in the event object's `coords` property. The listener callback will be executed every time your device detects a new location.

> ⚠ Continually monitoring the GPS for location will consume a mobile device's battery much faster than usual. Be sure that you actually need to be constantly handling the device's location before using this method. If you do, be sure to remove the `location` event listener via `Ti.Geolocation.removeEventListener()` when you are not actively using the location information.

Android lifecycle events

When monitoring location events continually in Android, apps will continue to receive events even when in the background. As mentioned above, this can be a major drain on the battery life of a mobile device. While this is sometimes the desired behavior, most apps only need location data while active.

In order to manage our location events such that we only receive them while our app is active, we need to take advantage of Titanium's access to the Android lifecycle events. There are three events of significance, each of which can be handled via `addEventListener()` on the `Ti.Android.currentActivity` object:

- `destroy` - This event is fired when your activity is destroyed. Location events should *always* be removed in this event.
- `pause` - This event is fired when an activity moves to the background. If you intend to suspend your location data handling when your app is in the background, you need to remove location event listeners in this event.
- `resume` - This event is fired when an activity comes to the foreground. If you previously removed location events in a `pause` event, this is where you would add them again to reenable them.

Below is a demonstration of how you would handle these events in order to only manage `location` events when your app is active. The key part to note is that pausing and resuming your `location` event handling is the responsibility of the Android Activity object accessible through the Titanium API as `Ti.Android.currentActivity`.

```
var locationAdded = false;
var handleLocation = function(e) {
    if (!e.error) {
        Ti.API.info(e.coords);
    }
};
var addHandler = function() {
    if (!locationAdded) {
        Ti.Geolocation.addEventListener('location', handleLocation);
        locationAdded = true;
    }
};
var removeHandler = function() {
    if (locationAdded) {
        Ti.Geolocation.removeEventListener('location', handleLocation);
        locationAdded = false;
    }
};

Ti.Geolocation.accuracy = Ti.Geolocation.ACCURACY_BEST;
Ti.Geolocation.preferredProvider = Ti.Geolocation.PROVIDER_GPS;
if (Ti.Geolocation.locationServicesEnabled) {
    addHandler();

    var activity = Ti.Android.currentActivity;
    activity.addEventListener('destroy', removeHandler);
    activity.addEventListener('pause', removeHandler);
    activity.addEventListener('resume', addHandler);
} else {
    alert('Please enable location services');
}
```

Use the device's compass

A mobile device's compass can be used to determine its heading. By using heading, the added dimension of direction can be added to a location based mobile app. With this addition, developers can add features like more robust navigation or even augmented reality.

Just as with location, Titanium has events and functions for both continual and one-time monitoring of heading. Also, check the API docs for platform-specific configuration information of heading options. For continual monitoring, the `heading` event needs to be registered with the `Ti.Geolocation` module. In the case of needing only the current heading, a simple call to the `Ti.Geolocation.getCurrentHeading()` function is necessary. As you may have noticed, this is very similar to how location is handled.

The below includes both of the methods for determining heading mentioned above.

```
if (Ti.Geolocation.locationServicesEnabled) {
    Ti.Geolocation.purpose = 'Get Current Heading';

    // make a single request for the current heading
    Ti.Geolocation.getCurrentHeading(function(e) {
        Ti.API.info(e.heading);
    });

    // Set 'heading' event for continual monitoring
    Ti.Geolocation.addEventListener('heading', function(e) {
        if (e.error) {
            alert('Error: ' + e.error);
        } else {
            Ti.API.info(e.heading);
        }
    });
} else {
    alert('Please enable location services');
}
```

The console output of your program will contain the heading information, which will be sent continuously from the `heading` event. The data for each heading entry will be structured in the following manner.

```
{
    "accuracy": 3,
    "magneticHeading": 34.421875,      // degrees east of magnetic north
    "timestamp": 1318447443692,
    "trueHeading": 43.595027923583984, // degrees east of true north
    "type": "heading",
    "x": 34.421875,
    "y": -69.296875,
    "z": -1.140625
}
```

Forward and reverse geocoding

Another feature of location services that is built into the Titanium API is geocoding. This is the process of converting an address into a geographic location (forward geocoding), or vice versa (reverse geocoding). For example, let's say we wanted to know the latitude and longitude of the Appcelerator headquarters in Mountain View, California. All we need to do is use the `Ti.Geolocation.forwardGeocoder()` function, giving it the address and a callback as parameters. Here's the code:

```
Ti.Geolocation.forwardGeocoder('440 Bernardo Ave Mountain View CA', function(e) {
    Ti.API.info(e);
});
```

And here is the output of a forward geocoding of Appcelerator HQ. As you can see, it delivers the geographic location of the given address in latitude and longitude.

```
{
    "accuracy": 1,
    "latitude": 37.389071,
    "longitude": -122.050156,
    "success": 1
}
```

Now let's say we just have latitude and longitude and we want to figure out what places of interest are in the area. This case can occur if you accept these coordinates from user input, or if you want to get further information in your `location` events. To do so, we use the `Ti.Geolocation.reverseGeocoder()` function. To this function we pass a latitude, longitude, and callback function. Let's see what we get when we use the random coordinates (50,50), as in the below sample.

```
Ti.Geolocation.reverseGeocoder(50, 50, function(e) {
    Ti.API.info(e);
});
```

Here's the output:

```
{
    "places": [
        {
            "address": ", 418020 Dzhany-Kuduk, , Kazakhstan",
            "city": "Oral",
            "country": "Kazakhstan",
            "country_code": "KZ",
            "latitude": 50.0,
            "longitude": 50.0,
            "street": "",
            "zipcode": 418020
        }
    ],
    "success": 1
}
```

While the above output shows only one place, you'll notice that the `places` property is an array. This means that on any given call to `Ti.Geolocation.reverseGeocoder()` you may receive a number of entries in the `places` property, if multiple places are found in the area of your query.

References

- W3C Geolocation API specification

Summary

In this chapter we learned how we can leverage a mobile device's native location services to add the context of a physical location to our apps. Using Titanium's APIs we are able to proactively query or passively listen for a device's current GPS position and heading. By using the configuration properties found in the `Ti.Geolocation` module like `accuracy` and `distanceFilter` we can further refine a location based experience.

Finally, we learned how to use additional location based features like forward and reverse geocoding to get even more location details. In the next chapter, we'll learn how we can use the native mapping functionality of mobile devices via the `Titanium.Maps` module. We'll be able to take the techniques learned in this chapter and apply them to the next in order to create a visual representation of our location data.