# Qt Mobility Usage Scenario: The mSense middleware

Example middleware that aggregates low level sensor information to provide more useful high level information.

**This article needs to be updated:** If you found this article useful, please fix the problems below then delete the {{ArticleNeedsUpdate}} template from the article to remove this warning.

**Reasons:** (13 Apr 2011)
Uses Mobile Extensions which are deprecated. Would be better to use Qt Mobility APIs. Also needs retesting against Qt SDK as older toolchain no longer supplied.

## Introduction

mSense is a novel context acquisition and management middleware for high-end mobile phones based on Qt. Within the middleware so called low-level sensor nodes encapsulate platform level APIs for seamlessly accessing hardware sensors, simulated sensor or web services, which are implicitly used as external data sources in the context of the middleware. High-level sensor nodes (so called channels or aggregators) combine information from other sensor nodes to acquire and extract particular sensor information and generate new knowledge. For example, GPS, an accelerometer and a compass could be combined to provide a more accurate inertial positioning sensor node. For the moment the output is supplied via a native interface, but some work on an additional broadcast feature is scheduled for the near future to enable publication of the output data through a network interface or a web service. The current version of the mSense middleware incorporates following low- and high-level sensor nodes:

- **Low-level sensor nodes**
  - *Accelerometer:* Sensor provides 3D acceleration of device at a configurable rate.
  - *Alarm Sensor:* Sensor provides information as well as notifications of current active alarms.
  - *Calendar Sensor:* Sensor provides current calendar data (meetings, etc.).
  - *Device Orientation:* Sensor provides current device orientation and attitude (rotation) information.
  - *GPS Location:* Sensor provides GPS based location information of the device.
  - *Magnetic North:* Sensor provides current heading of device with respect to magnetic north.
  - *Magnetometer:* Sensor provides 3D geomagnetic flux density information.
  - *Profile Sensor:* Sensor provides information about the current device profile.
  - *Weather Sensor:* Sensor provides actual weather information via a web-service for a configurable area.
- **High-level sensor nodes**
  - *User Availability:* Sensor provides information of the current user status (if the user is available or not). This is done by a combination of the calendar and device profile data.
  - *Movement Status:* Sensor provides information if the user is currently moving or not. This is done by a combination of the GPS location and the accelerometer sensor.
  - *Position/Address:* Sensor provides current position information of the user based on current GPS location. An additional a reverse geo-coder is used to retrieve the current position description (address data) for the GPS location.

For using those sensor nodes the mSense middleware provides a common high-level interface. In fact the individual data sources (hardware sensors, simulated sensors, device resources or web services) are encapsulated by the middleware, whereby internal access is implemented where applicable through modules of the Qt Mobility APIs [1] or available web interfaces. For the moment it should be noted that the mSense middleware additonally utilizes some temporary modules of the Qt Archived:Mobile Extensions project, since the current beta release of the Qt Mobility APIs does not provide functional backend implementations for all APIs (like for example the Sensor or the Calendar API [2]). This temporary backward compatibilities are intended to be removed and replaced by particular Qt Mobility API modules in the future.

For an additional overview of the mSense system architecture and the basic concepts behind mSense have a look at a dedicated paper on that topic from *Krösche et al.* [3].

## Build Instructions

Download the File:Msense-middleware-v1.3.2.zip and unzip it somewhere. If you intend using the mSense middleware within some of your projects it is a good approach to locate the middleware sources at the same root directory as your project. Move to that directory and build it with

- qmake
- make <cfg>

where <cfg> is a wildcard for your current build configuration and target (for example *debug-gcce*, *release-gcce* etc.)

- **General Information:** The mSense middleware is only available as dynamic library. This enables usage of the middleware in other projects and applications.
- **Project Dependencies:** Since the mSense middleware utilizes some modules of the new Qt APIs those Mobility APIs have to be installed and configured for the particular SDK properly. The necessary source and include files of the Qt Archived:Mobile Extensions dependency are directly included in the project's source tree.
- **Symbian Capabilities:** Due to the usage of the device's sensors, the location feature, network access and other user data and device readouts, the application needs at least *ReadDeviceData*, *WriteDeviceData*, *ReadUserData*, *WriteUserData*, *Location*, *Local Services*, *NetworkServices* and *UserEnvironment* capability.
- **Google Reverse Geo-Coding:** For reverse geo-coding (to get the position description or address data for the current GPS position) the Google Reverse Geo-Coding service is used. Therefore a Google API Key is needed. You have to register your own Google API Key [4]) and define it in the global definitions file for the reverse geo-coding sensor (in *src/sensors/reversegeocoder/ReverseGeocoderDefinitions.h*).

# Setup and using the mSense middleware

## Update include paths and project dependencies in your project file

In order to use the mSense middleware within your own project you have to setup the particular project dependencies and include path definitions in your project file. By default it is assumed that the source and include files of the middleware are located in a directory named *msense-middleware* at the same level as the new project's root directory.

```
INCLUDEPATH += ..\msense-middleware
```

Additionally it is assumed that the Qt Mobility APIs are installed and configured properly for your platform. If new to Qt and Qt Mobility see Getting started with Qt Mobility APIs for further details on this topic.

## Google API Key Definition

As already noted, you need a Google API key in order to use the reverse geo-coding sensor to get a position description or address information for a current GPS position via the *ReverseGeocoderSensor*. If not already done, you have to register this new Google API key [5]. The you have to define your key in the common definitions file for the reverse geo-coding sensor in *src/sensors/reversegeocoder/ReverseGeocoderDefinitions.h*'.

```
// FILE: reversegeocoderdefinitions.h

// Global Google API key used for web maps service access
static const QString GOOGLE_API_KEY = "Enter your key here";
```

## How to open a sensor node

To open a particular sensor node and get notified in case of changing sensor data or sensor status information two steps are neccessary. In the given example we are accessing the accelerometer low-level sensor via the mSense middleware to sense the device's current acceleration.

- First, you have to derive your class from the generic `ISensorProxy` or `IChannelProxy` interface respectively, depending whether you are monitoring a low-level (sensor) or high-level (channel) sensor node. This requires the implementation of some callback handlers which are actually used by the middleware for status and data notifications. At this point it should be noted that those callback functions of the interface are internally handled as Qt slots and notifications are processed via common signal and slot connections.

```
// FILE: accelerationreader.h

#include "src/global/MSenseInclude.h"

class AccelerationReader : public ISensorProxy
{

// ...

public:

    void onData(ISensor* aSrcSensor, TSensorDataPtr aData);

    void onSensorOpened(ISensor* aSrcSensor);

    void onSensorClosed(ISensor* aSrcSensor);

    void onSensorStatusChanged(ISensor *aSrcSensor, ISensor::ESensorStatus aStatus);

    void onSensorError(ISensor *aSrcSensor, ISensor::ESensorError aError);

// ...

};
```

- Secondly, you have to setup and open the sensor node based on a given sensor configuration as well as to implement the proxy callback overrides. The whole mSense middleware is accessed through the global `MSenseController` instance, which is provided as a singleton instance and manages the whole middleware.

```
// FILE: accelerationreader.cpp

#include "accelerationreader.h"
#include "src/global/SensorDefinitions.h"
#include "src/global/SensorConfigurations.h"


AccelerationReader::AccelerationReader()
{
    // Create sensor configuration for accleration sensor node
    TAccSensorConfig cfg;
    // Set an update interval of 100 ms (10 Hz) for the sensor
    cfg.p_updateInterval = 100;

    // Open the acceleration sensor. As soon as the sensor is open it will
    // start publishing it's data readings throught the sensor proxy interface
    ISensor::ESensorError err = MSenseController::instance()->openSensor(this, &cfg);

    if (ISensor::NoError == err)
    {
        // Sensor opened successfully...
    }
    else
    {
        // Failed to open sensor node (see particular error code)...
    }
}

void AccelerationReader::onData(ISensor* aSrcSensor, TSensorDataPtr aData)
```

```
{
    // Handle received sensor data...
}

void AccelerationReader::onSensorOpened(ISensor* aSrcSensor)
{
    // Handle sensor opened event...
}

void AccelerationReader::onSensorClosed(ISensor* aSrcSensor)
{
    // Handle sensor closed event...
}

void AccelerationReader::onSensorStatusChanged(ISensor *aSrcSensor, ISensor::ESensorStatus aStatus)
{
    // Handle sensor status changed event...
}

void AccelerationReader::onSensorError(ISensor *aSrcSensor, ISensor::ESensorError aError)
{
    // Handle sensor data error. Usually close the sensor and try to re-open.
}
```

### How to access data of a particular sensor node

All data readings of both low-level and high-level sensor nodes are published as a generic data container instance which provides access to concrete data attribute values through unique IDs. These IDs are defined for each sensor node in a separate definitions include file (see for example `AccDefinitions.h`).

```
// FILE: accelerationreader.cpp

#include "accelerationreader.h"
#include "src/global/SensorDefinitions.h"
#include "src/global/SensorConfigurations.h"

// ...

void AccelerationReader::onData(ISensor* aSrcSensor, TSensorDataPtr aData)
{
    if (aSrcSensor->getSensorID() != eSensorID_Accelerometer)
        return;

    int x = aData->intValue(eAccSensorAtt_XAxis);
    int y = aData->intValue(eAccSensorAtt_YAxis);
    int z = aData->intValue(eAccSensorAtt_ZAxis);

    // TODO: Process acceleration readings...
}

// ...
```

## mSense Demo-Application

### Introduction

The mSense Demo-Application is a simple demo and showcase application built upon the mSense middleware and realizes more or less a proof-of-concept of the proposed context acquisition approach. The application utilizes the middleware to acquire, visualize and process the data available from real or simulated low-level sensor nodes (like for example the accelerometer, magnetic compass, or other sensors) as well as from high-level sensors (like for example a reverse geo-coder based on current GPS location information etc.). In fact there is a short demo scenario available for each sensor node supported by the mSense middleware. Therefore the main purpose of this application is still the demonstration of the mSense Middleware and it’s usage within other projects or applications.

Another possible approach is to acquire context data within the the mSense middleware and publish or integrate this information (Movement, Position- and User Availability data) into the Facebook Platform Community. Therefore the so called Facebook Connect approach is used.


## Build Instructions

Download the source code for the demo application File:Msense-demo-application-v1.3.2.zip, unzip it in the same directory as the mSense middleware and check the following dependencies.

- **Project Dependencies:** Due to the dependency to the mSense middleware the particular project include paths have to be properly adopted (by default it is assumed that the demo application's root directory is at the same level as the mSense middleware root directory).
- **Symbian Capabilities:** The demo application needs at least the same capabilities as themSense middleware – *ReadDeviceData*, *WriteDeviceData*, *ReadUserData*, *WriteUserData*, *Location*, *NetworkServices*, *LocalServices* and *UserEnvironment*.
- **Facebook Connect:** For the Facebook integration feature an application key and also an application secret key is needed. Therefore you have to create and register a new Facebook application directly on the facebook platform [6]. The specific keys need to be defined in the global key file *src/api_keys.h*.
- **Google Reverse Geo-Coding and Google Maps:** To request a map of the current location via Google Maps a Google API key [7] is required. In case of the demo application the key from the mSense middleware could be used and has to be defined in the global key file *src/api_keys.h*.

```
// FILE: api_keys.h

// Facebook API keys ...
static const QString FACEBOOK_APP_KEY = "Enter your app key here";
static const QString FACEBOOK_SECRET_KEY = "Enter your secret key here";

// Global Google API key used for web maps service access
static const QString GOOGLE_API_KEY = "Enter your key here";
```

If all dependencies are fixed, move to the demo application's root directory and build it with

- qmake
- make <cfg>

where <cfg> is a wildcard for your current build configuration and target (for example *debug-gcce*, *release-gcce* etc.)

## Using the mSense Demo-Application

In the *General Settings* Screen it is possible to setup the configuration of the demo application. If you want to share your current context information on Facebook, you first need to login to the platform. This can be done with any registered Facebook account. After this you can start sensing context information (within the current example we will sense the movement status, actual position information and availability data based on calendar and phone profile settings). Concerning the position information you can either use a simulated location or the actual real position as provided by the device's GPS receiver. To finally share the personal context information, click the *FB Sharing* Button. A preview with your current context data is shown, containing

- your position information (also integrated in a Google Maps image),
- your movement status with the current speed,
- your availability data (like for example: appointment from 14:45 till 15:30 with particular description from the calendar).

It is also possible to add a personal comment to the current context information in the text field above. By pressing the Publish Button your personal status is shared on your Facebook account.