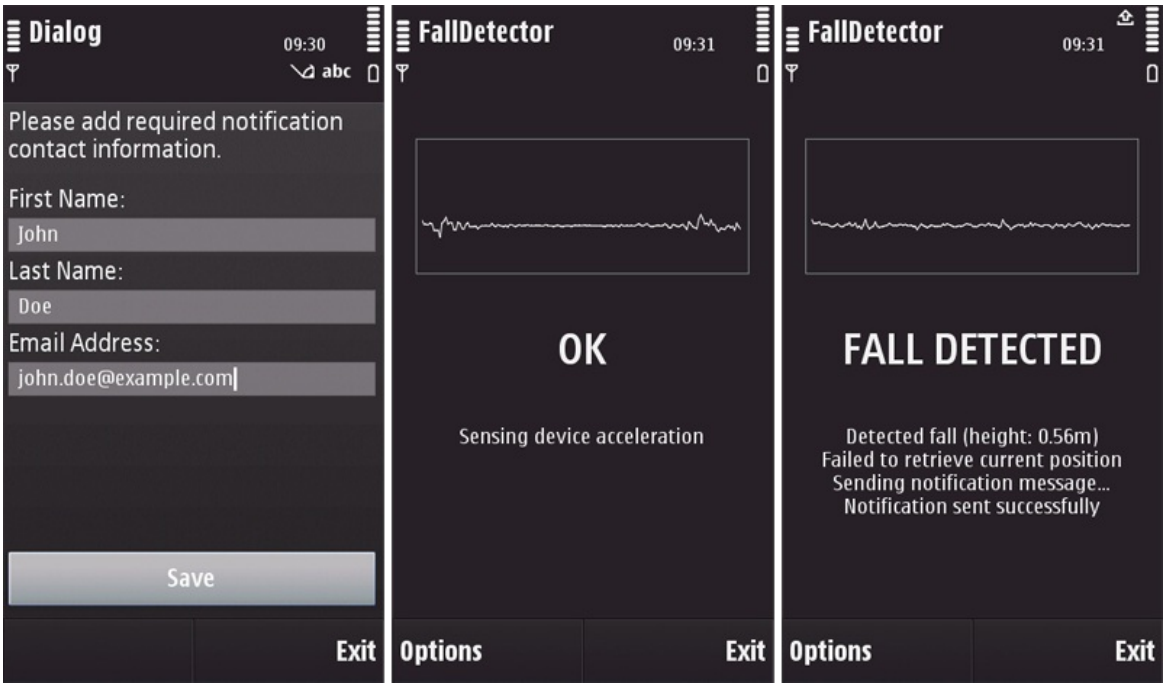
Qt Mobility example application: Fall Detector

The Fall Detector demo application automatically detects when a person falls down and provides a simple notification facility for detected downfalls (e.g. to request first aid etc.). In general the given example application demonstrates the utilization of Sensor, Location, Contacts and Messaging modules from the Qt Mobility APIs Beta[1] release



Build Instructions

Download the File:Falldetector.zip and unzip it somewhere. 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.)

Capabilities

The application requires the following capabilities:

CAPABILITY = Location NetworkServices LocalServices ReadUserData WriteUserData UserEnvironment ReadDeviceData WriteDeviceData

Using the fall detector application

- Start the fall detector application
- If the application is started for the first time, you will be prompted to enter some notification details (name and email address). This details will be saved in your local contact store. In case of a detected fall the application uses the given email address for notification as well.

Code Snippets

Using the Sensor API to get current device acceleration

```
#include <QAccelerometer>

// Neccessary for Qt Mobility API usage
QTM_USE_NAMESPACE

class AccelerationInfo : public QObject, public QAccelerometerFilter
{
    Q_OBJECT

public:

    AccelerationInfo(QObject* parent = 0) : QObject(parent)
    {
        m_sensor = new QAccelerometer(this);
        m_sensor->addFilter(this);
        m_sensor->start();
    }
```


(C) Copyright Nokia 2013. All rights reserved.

```
private slots:

    // Override of QAccelerometerFilter::filter(QAccelerometerReading*)
    void filter(QAccelerometerReading* reading)
    {
        qreal x = reading->x();
        qreal y = reading->y();
        qreal z = reading->z();

        // Process acceleration sensor readings ...

        qDebug("Current device acceleration: x=%f y=%f z=%f", x, y, z);
    }

private:

    QAccelerometer* m_sensor;
};
```

Using the Location API to get current position

```
#include <QGeoPositionInfo>
#include <QGeoPositionInfoSource>

// Neccessary for Qt Mobility API usage
QTM_USE_NAMESPACE

class LocationInfo : public QObject
{
    Q_OBJECT

public:

    LocationInfo(QObject* parent = 0) : QObject(parent)
    {
        QGeoPositionInfoSource* src = QGeoPositionInfoSource::createDefaultSource(this);
        if (src)
        {
            connect(src, SIGNAL(positionUpdated(QGeoPositionInfo)), this,
                SLOT(updatePosition(QGeoPositionInfo));
            connect(src, SIGNAL(updateTimeout()), this, SLOT(updateTimeout()));
            src->requestUpdate(5000); // Start request for actual position
        }
    }

private slots:

    void updatePosition(const QGeoPositionInfo& info)
    {
        qDebug() << “Current position: ” << info;
    }

    void updateTimeout()
    {
        // Current location could not be retrieved within the specified timeout of 5 seconds.
        qWarning(“Failed to retrieve current position.”);
    }
};
```

Using the Contacts API

Create an example contact and populate it with data

```
// Construct contact manager for default contact backend
QContactManager* cm = new QContactManager();
// Create example contact
QContact example;

// Add contact name
QContactName name;
name.setFirst(“John”);
name.setLast(“Doe”);
example.saveDetail(&name);

// Add contact email address
QContactEmailAddress email;
email.setContexts(QContactDetail::ContextHome);
email.setEmailAddress(“john.doe@example.com”);
example.saveDetail(&email);

// Finally, save the contact details
cm->saveContact(&example);
```

Add custom field *Interests* to a note detail associated with a contact

```
// Get the contact definition we want to modify (cm is a
// previously created QContactManager object)
QMap<QString, QContactDetailDefinition> def = cm->detailDefinitions();
QContactDetailDefinition modified = def.value(QContactNote::DefinitionName);

// Define new field ‘Interests’
QContactDetailDefinitionField field;
field.setDataType(QVariant::String);
QMap<QString, QContactDetailDefinitionField> fields = modified.fields();

// Insert new field to the list of contact’s fields and update
// the contact’s field definitions
fields.insert(“Interests”, field);
modified.setFields(fields);

// Save the updated field definitions back to the manager
cm->saveDetailDefinition(modified);
```

Search for a contact based on an incoming call ID

(C) Copyright Nokia 2013. All rights reserved.

```
// Create new contact detail filter
QContactDetailFilter filter;
filter.setDetailDefinitionName(QContactPhoneNumber::DefinitionName,
    QContactPhoneNumber::FieldNumber);

// Specify the requested value and filter matching criteria
QString incomingNbr("+436641234567");
filter.setValue(incomingNbr);
filter.setMatchFlags(QContactFilter::MatchExactly);

// Fetch matching contacts list for specified filter (cm is a
// previously created QContactManager object)
QList<QContactLocalId> matches = cm->contacts(filter);

for (int i = 0; i < matches.size(); ++i)
    qDebug() << cm->contact(matches.at(i));
```

Using the Message API to send an email message with a photo attachment

```
// Setup new email message
QMessage msg;
msg.setType(QMessage::Email);

// Set recipient for our email message
QString recipient("john.doe@example.com");
msg.setTo(QMessageAddress(QMessageAddress::Email, recipient));

// Define message subject, body and append attachment
msg.setSubject("Messaging API example");
msg.setBody("Hello,\n\nthis is an example message.");

QStringList attachments;
attachments << "images/example.png";
msg.appendAttachments(attachments);

// Send message using a new service handle
QMessageService* svc = new QMessageService();

if (svc->send(msg))
    qDebug("Successfully sent message.");
else
    qWarning("Failed to send message.");
```


(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.