1. Introduction

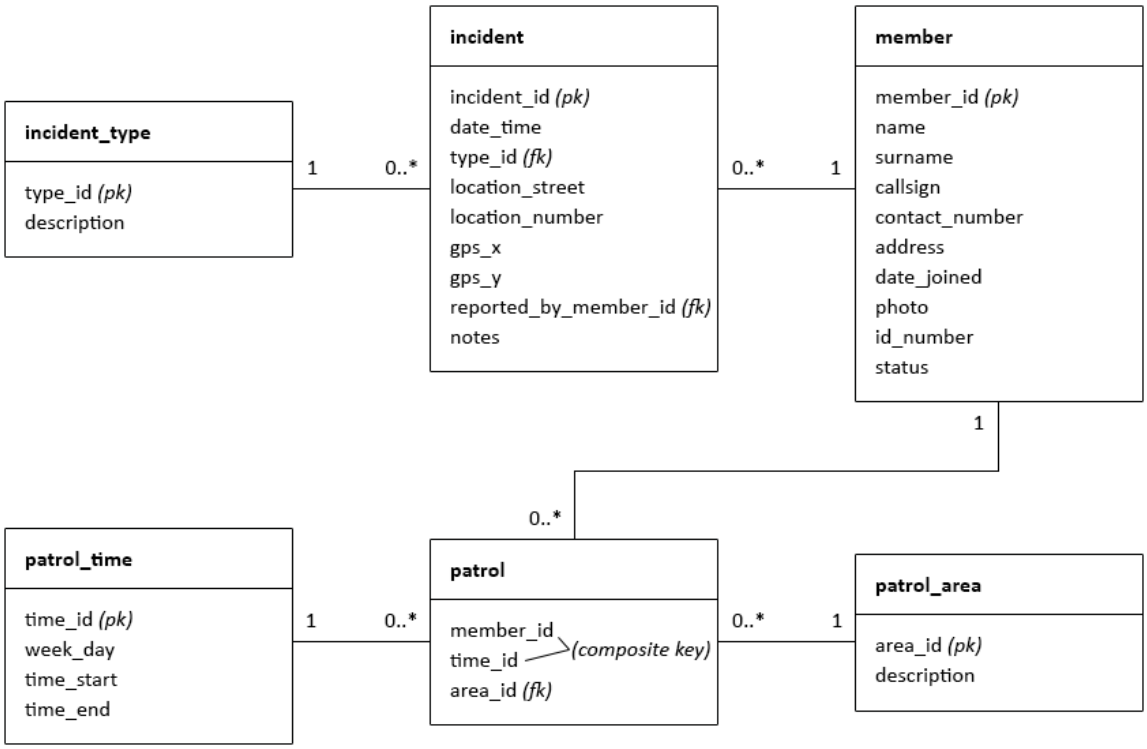
My application is the *TVNW Incident Reporting System* for use by the members of the *Table View Neighbourhood Watch*. The neighbourhood watch patrols the surburb looking for suspicious activity and members remain in contact with others on patrol to coordinate surveillance on suspected criminals' movements until the police can arrive to take over. If the details of all these incidents are recorded, it could reveal some patterns or hot spots which would lead to more effective law enforcement deployment. This application aims to make the recording of these incidents easier, and also to help coordinate more effective patrolling by members.

Features:

- Manage the details and patrols of members of the watch;
- See who is on patrol at the current time and their contact details;
- Manage patrol time slots and areas that members can be linked to;
- Record the type and details of any suspicious or criminal incidents that occur in the area.

2. Database design specifications

The database has six tables: `incident`, `incident_type`, `member`, `patrol`, `patrol_time` and `patrol_area`. The `incident` table is related to the `member` table in that an incident is reported by a member. A member can report many incidents. A member can also have many `patrol`s which are related to the `patrol_time` table, where the available time slots are defined, and the `patrol_area` table, where suburb zones are defined. This is the database UML diagram below. The fields and data types are on the following page.



MySQL 5.7 Command Line Client

```
mysql> show tables;
+----------------+
| Tables_in_tvnw |
+----------------+
| incident       |
| incident_type  |
| member         |
| patrol         |
| patrol_area    |
| patrol_time    |
+----------------+
6 rows in set (0.00 sec)

mysql> explain incident;
+-----------------------+--------------+------+-----+---------+----------------+
| Field                 | Type         | Null | Key | Default | Extra          |
+-----------------------+--------------+------+-----+---------+----------------+
| incident_id           | int(11)      | NO   | PRI | NULL    | auto_increment |
| date_time             | datetime     | NO   |     | NULL    |                |
| type_id               | int(11)      | NO   | MUL | NULL    |                |
| location_street       | varchar(100) | NO   |     | NULL    |                |
| location_number       | smallint(6)  | NO   |     | NULL    |                |
| gps_x                 | double       | YES  |     | NULL    |                |
| gps_y                 | double       | YES  |     | NULL    |                |
| reported_by_member_id | int(11)      | NO   | MUL | NULL    |                |
| notes                 | text         | YES  |     | NULL    |                |
+-----------------------+--------------+------+-----+---------+----------------+
9 rows in set (0.00 sec)

mysql> explain incident_type;
+-------------+--------------+------+-----+---------+----------------+
| Field       | Type         | Null | Key | Default | Extra          |
+-------------+--------------+------+-----+---------+----------------+
| type_id     | int(11)      | NO   | PRI | NULL    | auto_increment |
| description | varchar(255) | NO   |     | NULL    |                |
+-------------+--------------+------+-----+---------+----------------+
2 rows in set (0.00 sec)

mysql> explain member;
+----------------+--------------+------+-----+---------+----------------+
| Field          | Type         | Null | Key | Default | Extra          |
+----------------+--------------+------+-----+---------+----------------+
| member_id      | int(11)      | NO   | PRI | NULL    | auto_increment |
| name           | varchar(100) | NO   |     | NULL    |                |
| surname        | varchar(100) | NO   |     | NULL    |                |
| callsign       | varchar(50)  | NO   |     | NULL    |                |
| contact_number | int(11)      | NO   |     | NULL    |                |
| address        | varchar(255) | YES  |     | NULL    |                |
| date_joined    | date         | NO   |     | NULL    |                |
| photo          | varchar(255) | YES  |     | NULL    |                |
| id_number      | bigint(20)   | NO   |     | NULL    |                |
| status         | tinyint(1)   | NO   |     | NULL    |                |
+----------------+--------------+------+-----+---------+----------------+
10 rows in set (0.00 sec)
```

MySQL 5.7 Command Line Client

```
mysql> explain patrol;
+-----------+---------+------+-----+---------+-------+
| Field     | Type    | Null | Key | Default | Extra |
+-----------+---------+------+-----+---------+-------+
| member_id | int(11) | NO   | PRI | NULL    |       |
| time_id   | int(11) | NO   | PRI | NULL    |       |
| area_id   | int(11) | NO   | MUL | NULL    |       |
+-----------+---------+------+-----+---------+-------+
3 rows in set (0.00 sec)

mysql> explain patrol_area;
+-------------+--------------+------+-----+---------+----------------+
| Field       | Type         | Null | Key | Default | Extra          |
+-------------+--------------+------+-----+---------+----------------+
| area_id     | int(11)      | NO   | PRI | NULL    | auto_increment |
| description | varchar(100) | NO   |     | NULL    |                |
+-------------+--------------+------+-----+---------+----------------+
2 rows in set (0.00 sec)

mysql> explain patrol_time;
+------------+-------------+------+-----+---------+----------------+
| Field      | Type        | Null | Key | Default | Extra          |
+------------+-------------+------+-----+---------+----------------+
| time_id    | int(11)     | NO   | PRI | NULL    | auto_increment |
| week_day   | varchar(50) | NO   |     | NULL    |                |
| time_start | time        | NO   |     | NULL    |                |
| time_end   | time        | NO   |     | NULL    |                |
+------------+-------------+------+-----+---------+----------------+
4 rows in set (0.00 sec)
```

I also created a view on the tables called `linked_patrols` while developing the application. I used the following query to do it.

```
CREATE VIEW linked_patrols AS SELECT member_id, patrol_time.time_id,
patrol_area.area_id, patrol_time.week_day, patrol_time.time_start,
patrol_time.time_end, patrol_area.description FROM patrol INNER JOIN patrol_time
ON patrol.time_id=patrol_time.time_id INNER JOIN patrol_area ON
patrol.area_id=patrol_area.area_id;
```

MySQL 5.7 Command Line Client

```
mysql> explain linked_patrols;
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| member_id   | int(11)      | NO   |     | NULL    |       |
| time_id     | int(11)      | NO   |     | 0       |       |
| area_id     | int(11)      | NO   |     | 0       |       |
| week_day    | varchar(50)  | NO   |     | NULL    |       |
| time_start  | time         | NO   |     | NULL    |       |
| time_end    | time         | NO   |     | NULL    |       |
| description | varchar(100) | NO   |     | NULL    |       |
+-------------+--------------+------+-----+---------+-------+
7 rows in set (0.00 sec)
```

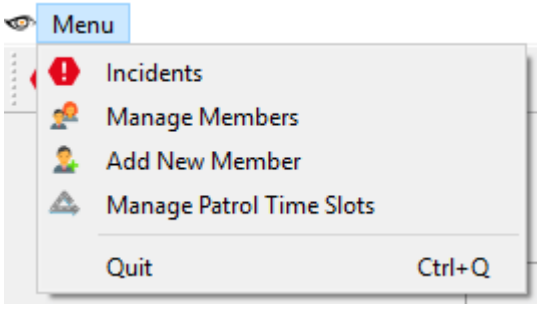
3. User interface design

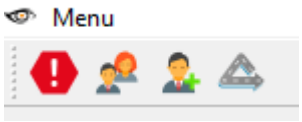
I will break up this section into explanations of each of the application's windows as follows:

i. **Adding a new member**
ii. **Managing patrol time slots**
    a. Adding a new patrol time slot
    b. Removing a patrol time slot
iii. **Managing members**
    a. Updating personal details
    b. Adding and removing linked patrols
    c. Find member by name
iv. **Managing incidents**
    a. Add new incident
    b. Update or delete incident
    c. Filter by date range
    d. See currently patrolling members

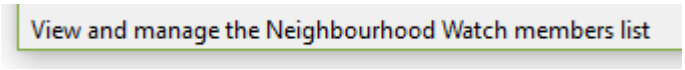
Brief description of navigation and list of classes and widgets used

I'm using QMenuBar, QMenu and QToolBar which are connected to QActions as the means of navigation between the four subwindows of a QMdiArea.





I also implemented QStatusBar to show tips when hovering over the menu items.

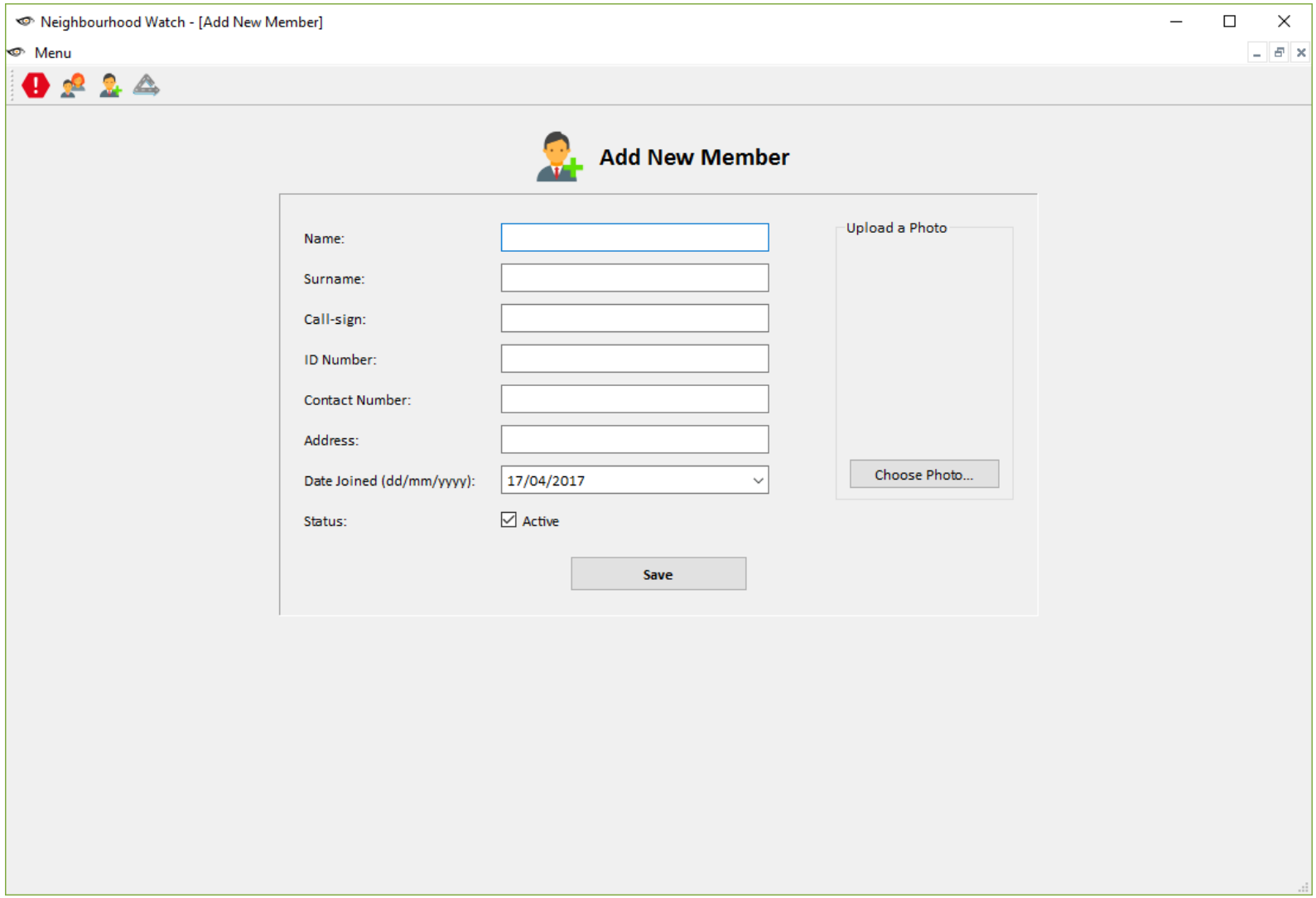


I'll mention some more classes and widgets on the specific windows, but here is a list too:

- QMainWindow, QWidget, QMdiArea, QTabWidget
- QGridLayout, QFrame, QGroupBox, QHBoxLayout, Spacer
- QLabel, QLineEdit, QTextEdit, QDateEdit, QDateTimeEdit, QComboBox, QCheckBox, QPushButton
- QListWidget, QTableView, QWebView, QLCDNumber
- QFileDialog, QMessageBox, QTimer, QTime, QDate, QDateTime, QPixmap

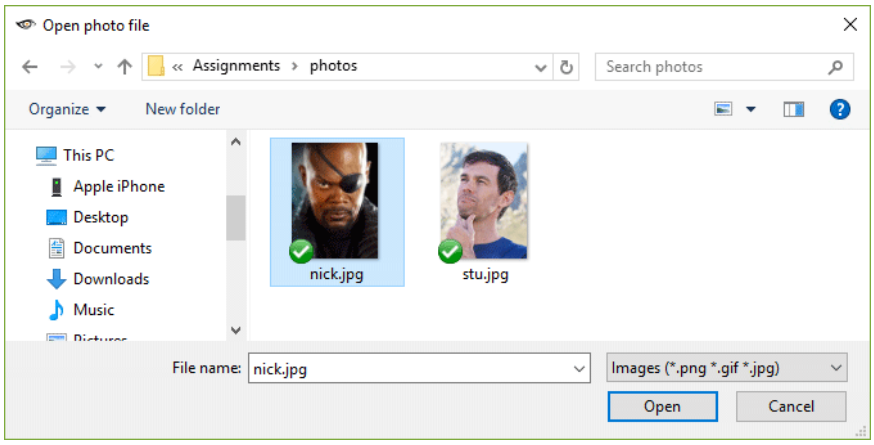
3. i. Adding a new member

This is the third window of the application and a good starting point for explanation.

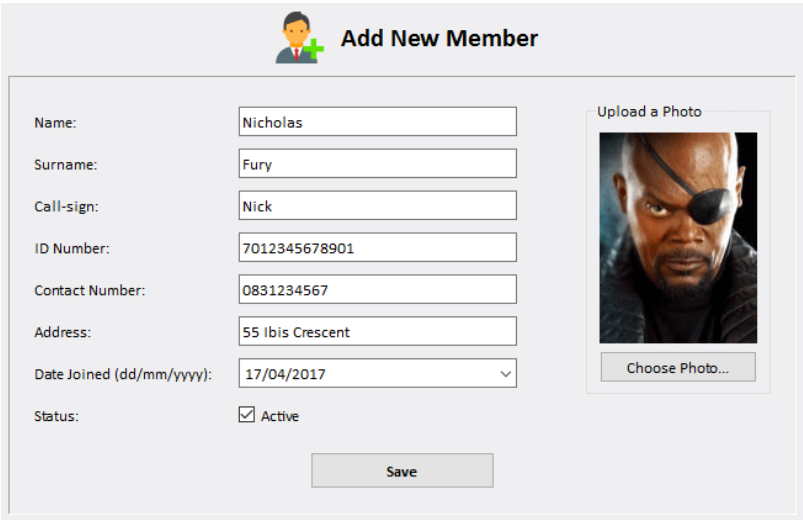


When a new member wants to join the Neighbourhood Watch, their details can be captured on this screen. The 'Date Joined' QDateEdit is populated by the current date automatically, and the QCheckBox is checked by default.

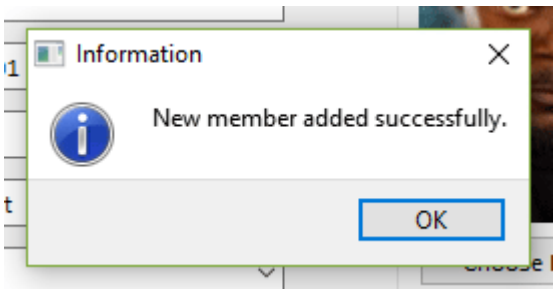
At this point a photo can also be added to the member's profile. Clicking the 'Choose Photo...' QPushButton will bring up a QFileDialog where the user can choose an image:



I use QPixmap to show the image in a QLabel. A completed form looks like this:

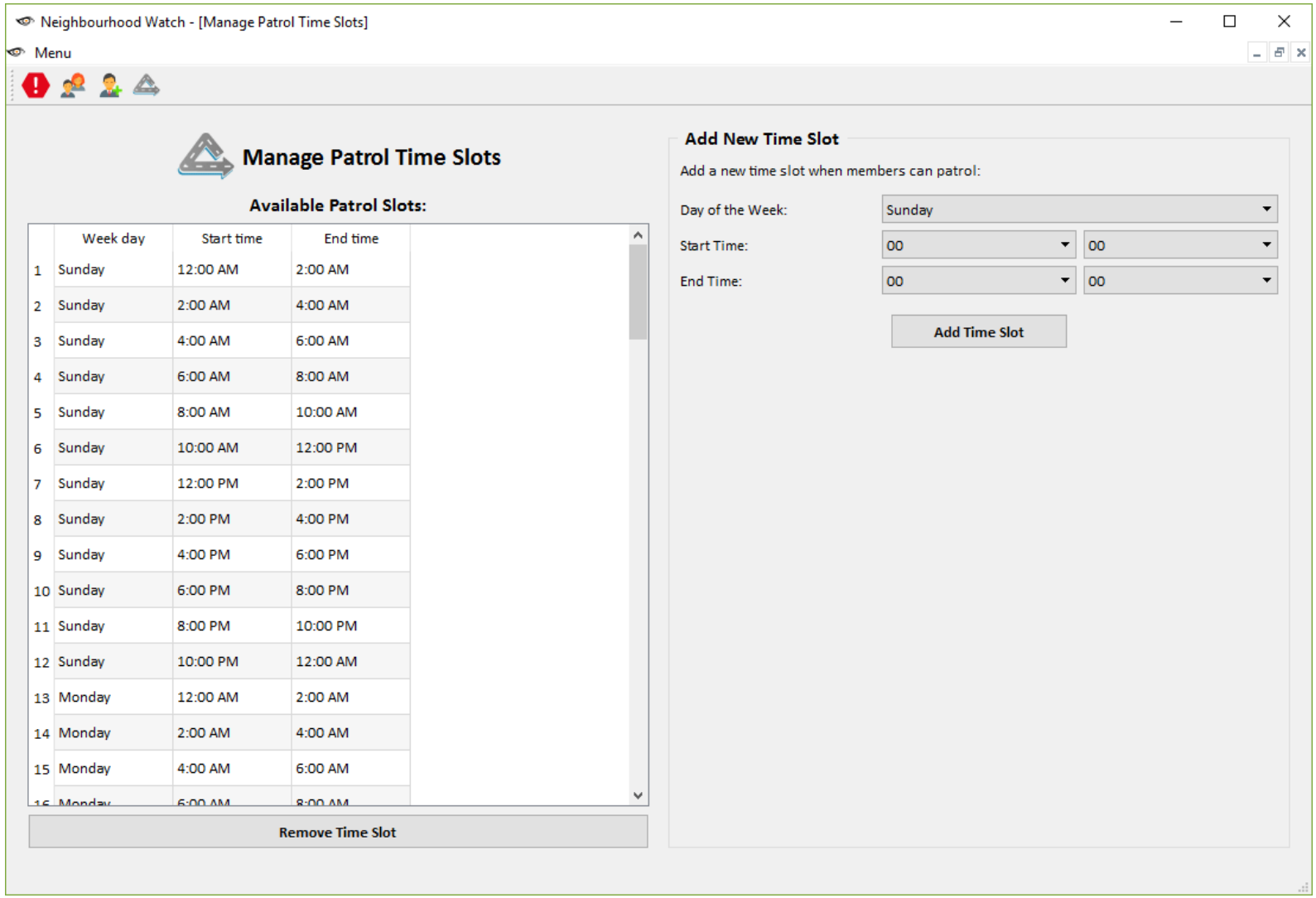


Once you hit 'Save', the details are validated and written to the database (code provided later) and a QMessageBox is shown to the user, after which all the form fields are reset. The member QTableView is also updated in the 'Manage Members' window (see page 9).



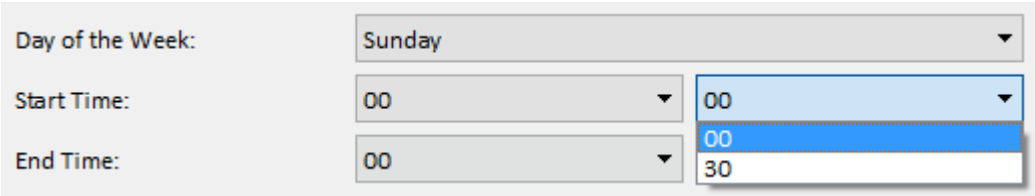
3. ii. Managing patrol time slots

In this window, time slots can be added or removed. I've added twelve 2-hour slots per day of the week in my example.

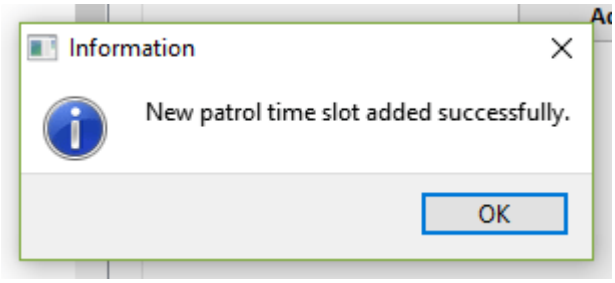


3. ii. a. Adding a new patrol time slot

The QComboBoxes are pre-populated in PyQt with the days of the week, the 24 hours of the day in the left combo boxes (starting from midnight) and two options in each of the right combo boxes, 00 and 30. The shortest a slot can be is 1 hour, and thereafter in increments of 30 minutes.

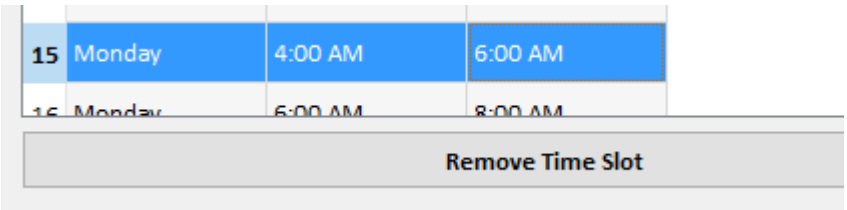


When the user hits 'Add Time Slot', some validation is done on the combo boxes, as well as on the database to check whether it already exists. On successful insertion, a QMessageBox is shown and the 'Available Patrol Slots' QTableView is updated. The list of available time slots is also updated on the 'Manage Members' window (see next section).

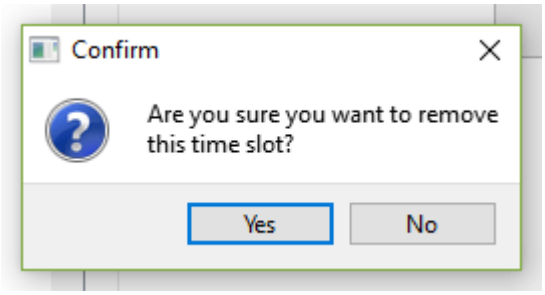


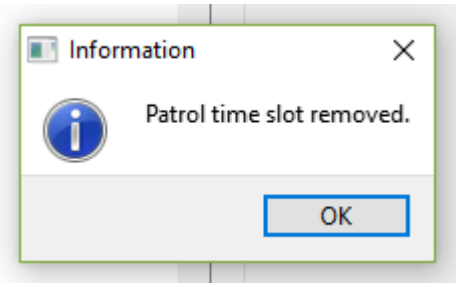
3. ii. a. Removing a patrol time slot

A patrol time slot can be removed by selecting a table row, and clicking 'Remove Time Slot'. The QTableView has been set to *SingleSelection* and *SelectedRows* in PyQt (all the tables in this application behave this way), thus allowing only 1 row to be selected at at time.



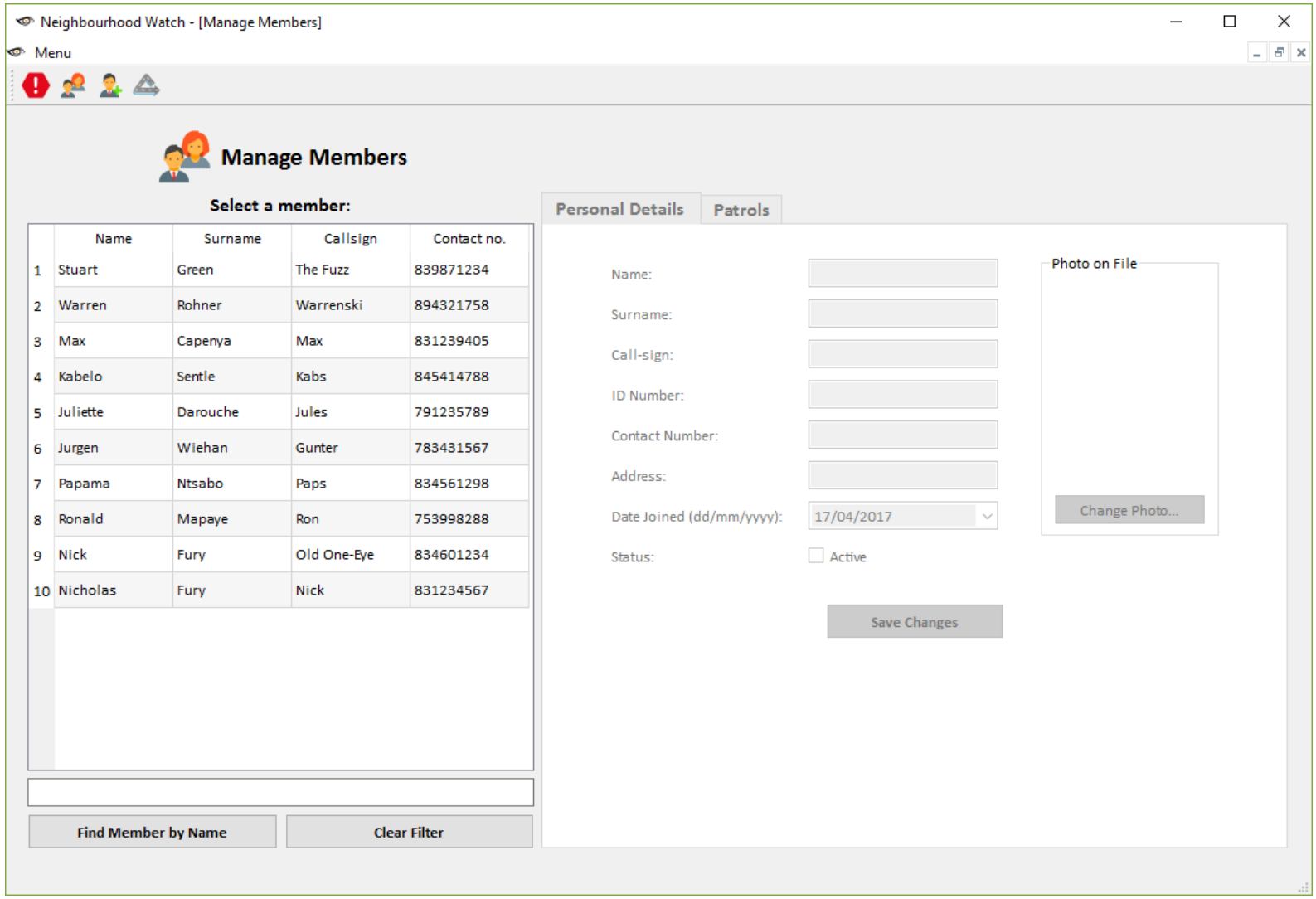
A confirmation QMessageBox is shown to the user, followed by another confirming the deletion.





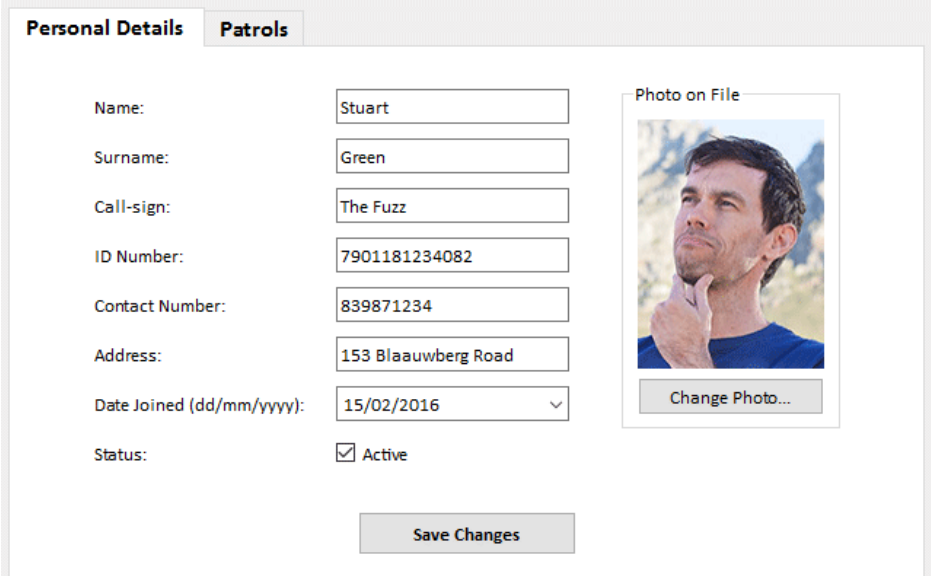
3. iii. Managing members

In this window, member details and member patrols can be managed. The tabs are initially disabled until a member is selected from the QTableView.



3. iii. a. Updating personal details

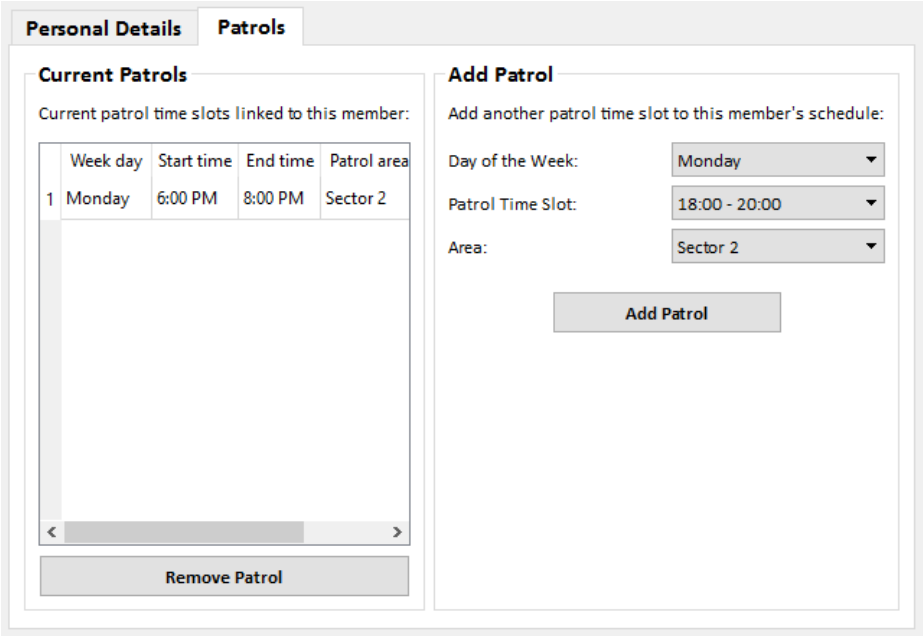
Selecting a member from the QTableView will enable the QTabWidget and populate both tabs with that member's details and patrol information.



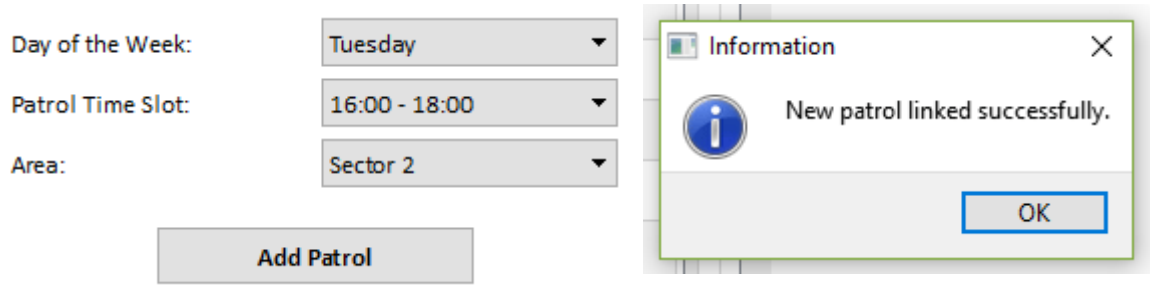
Here the user can modify the details, change the photo (same functionality as the 'Add New Member' window – see page 6), and hit 'Save Changes' to write the changes to the database. The same validation is done as the 'Add New Member' window and a QMessageBox is shown to confirm the insertion.

3. iii. b. Adding and removing linked patrol

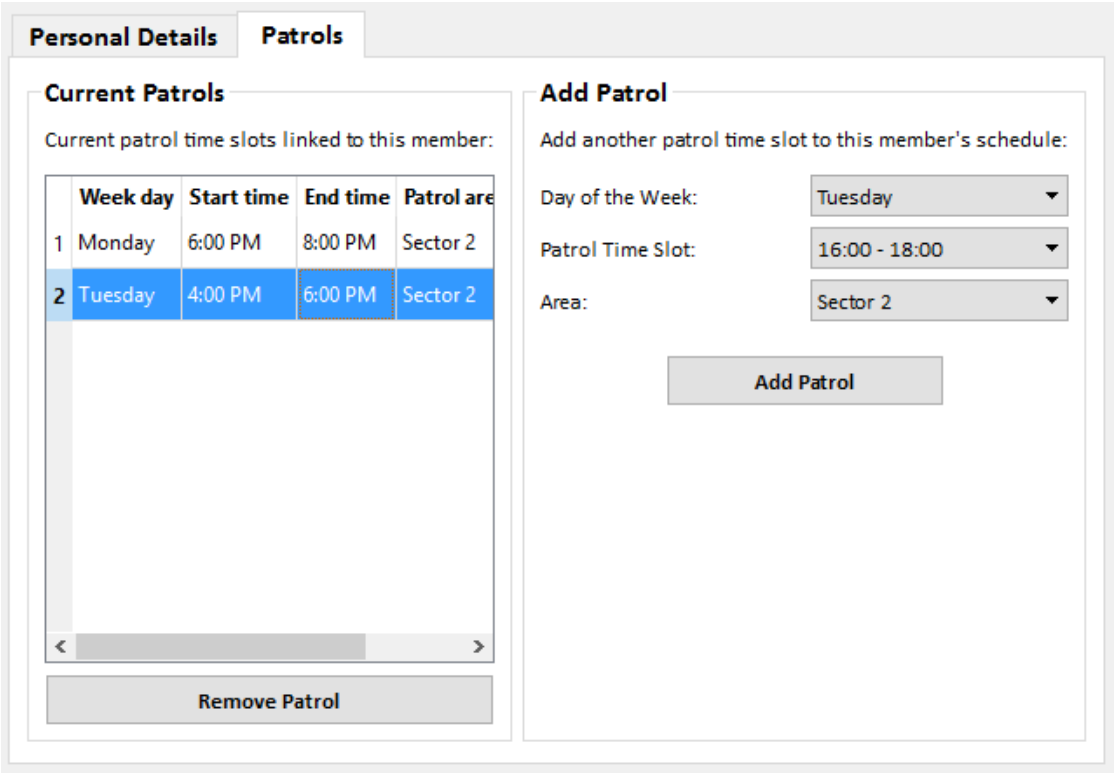
On the 'Patrols' tab for this member, you can see what times the member patrols, and link more patrol times to the member. The QComboBoxes for 'Patrol Time Slot' and 'Area' are populated from the database.



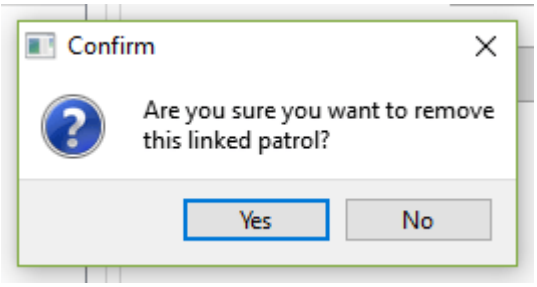
For example, making the following selection and hitting 'Add Patrol' will add that patrol to the member's patrols and refresh the 'Current Patrols' QTableView:

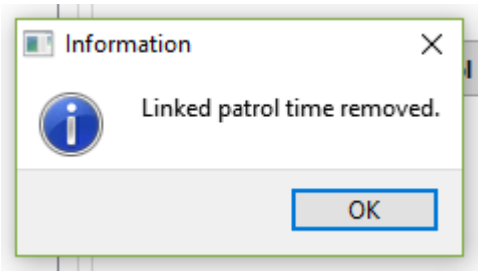


The new patrol is now listed in the table:



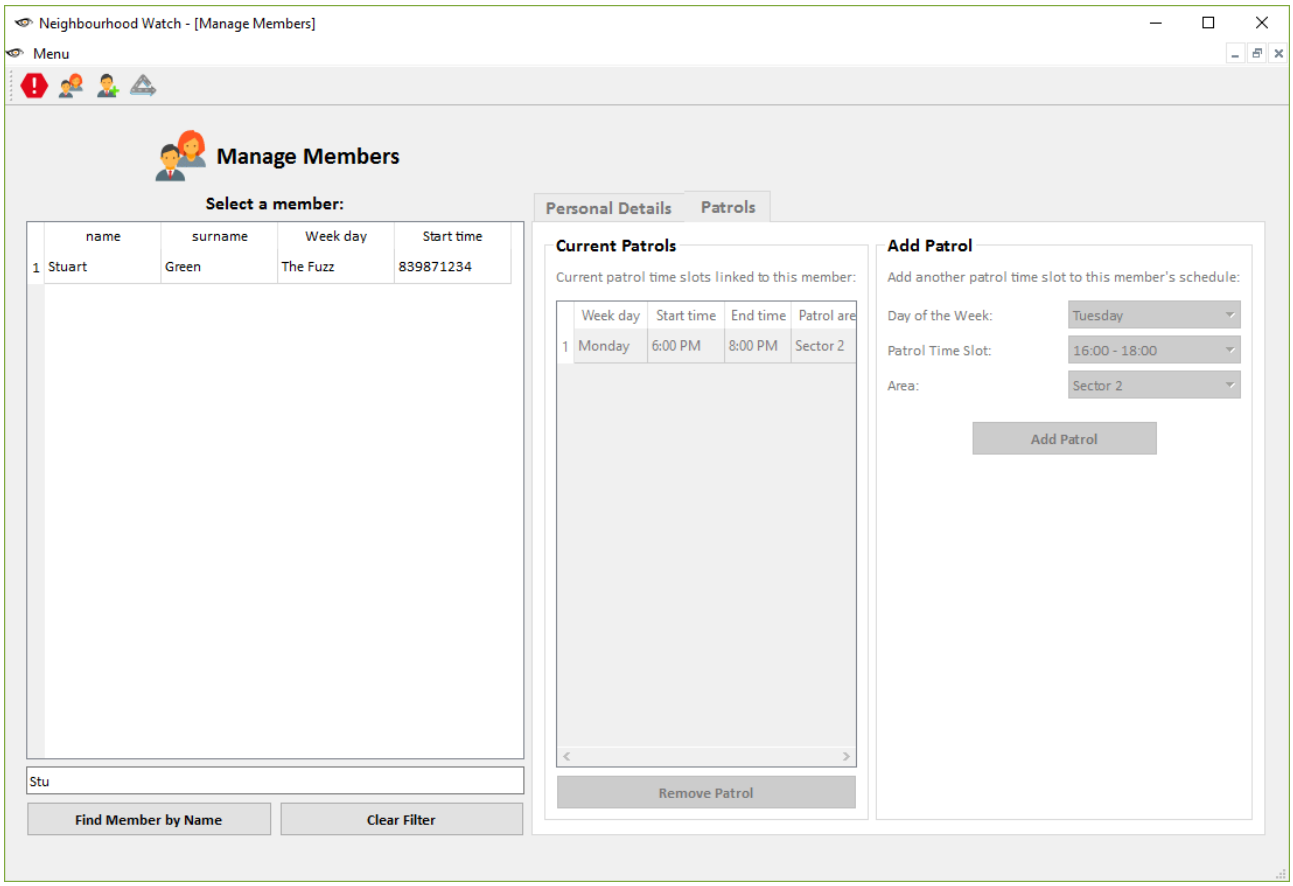
Selecting a patrol in the table and clicking 'Remove Patrol' will open a QMessageBox for confirmation and then confirm the deletion as well.





3. iii. c. Find member by name

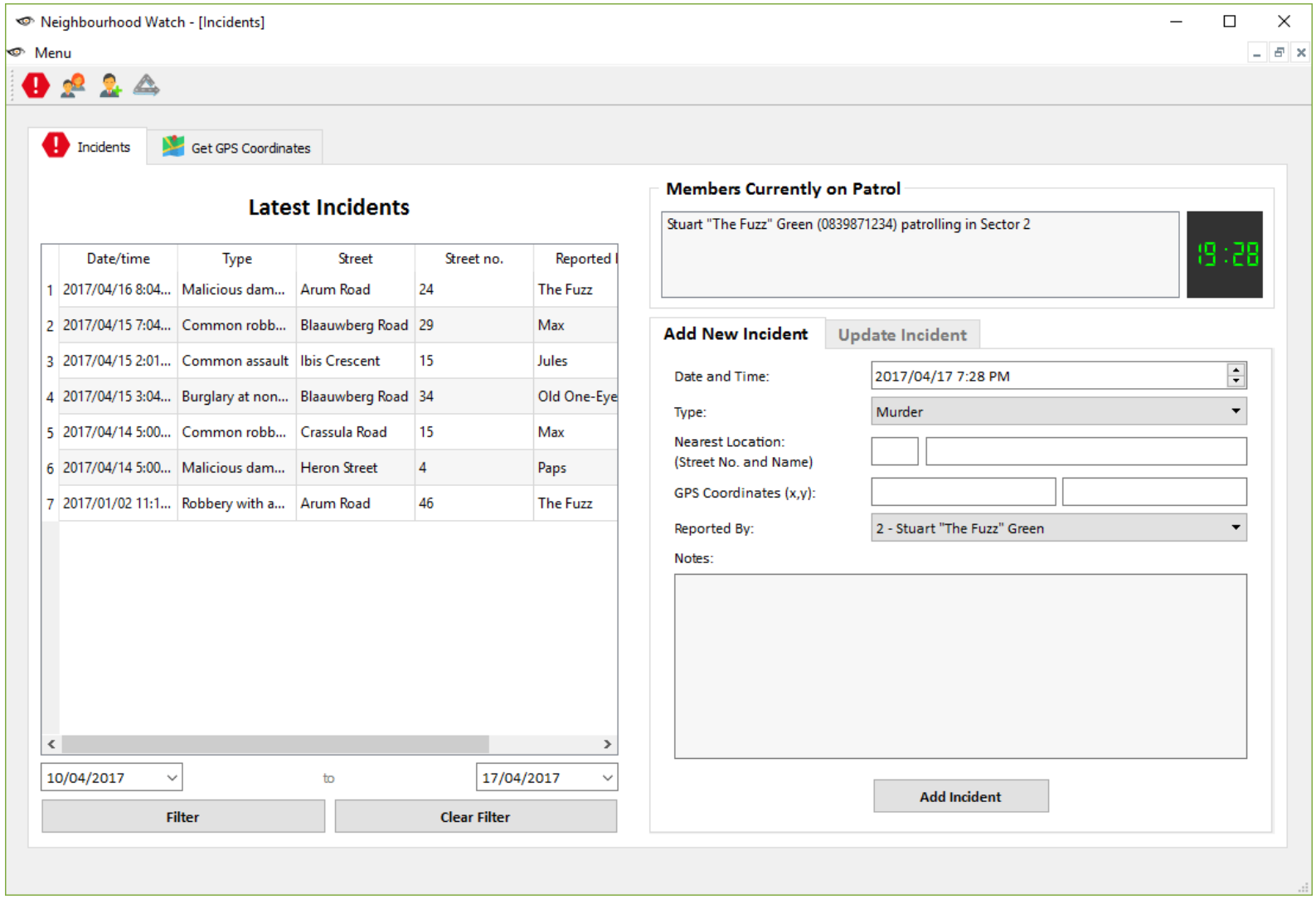
You can also apply a filter to the QTableView based on the member's first name. Type a name (or part of a name) into the QLineEdit and hit 'Find Member by Name' to show matching results in the table. This also disables the QTabWidget since no member is selected anymore.



Clicking on 'Clear Filter' resets the QLineEdit and reloads the QTableView with all the members listed again.

3. iv. Managing incidents

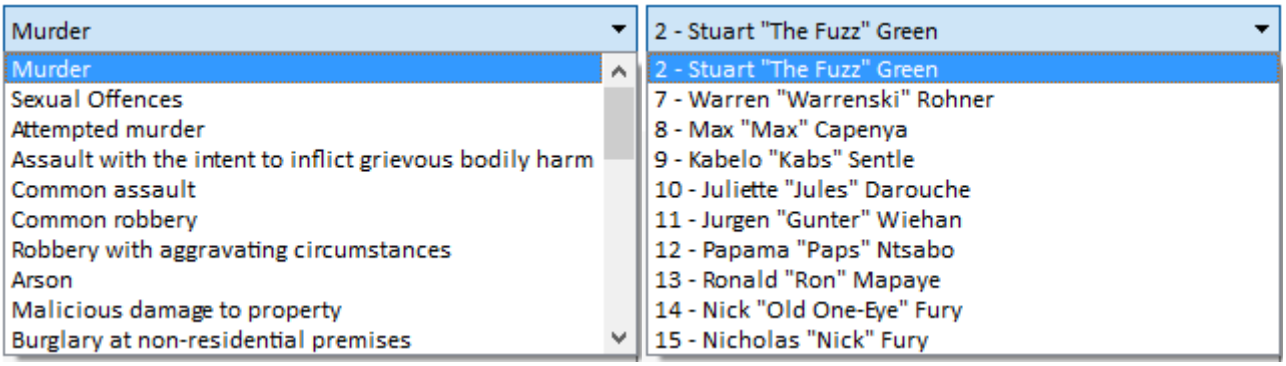
This is the window that the application starts up in. It's the window where incidents are listed, recorded and updated, and other information is shown.



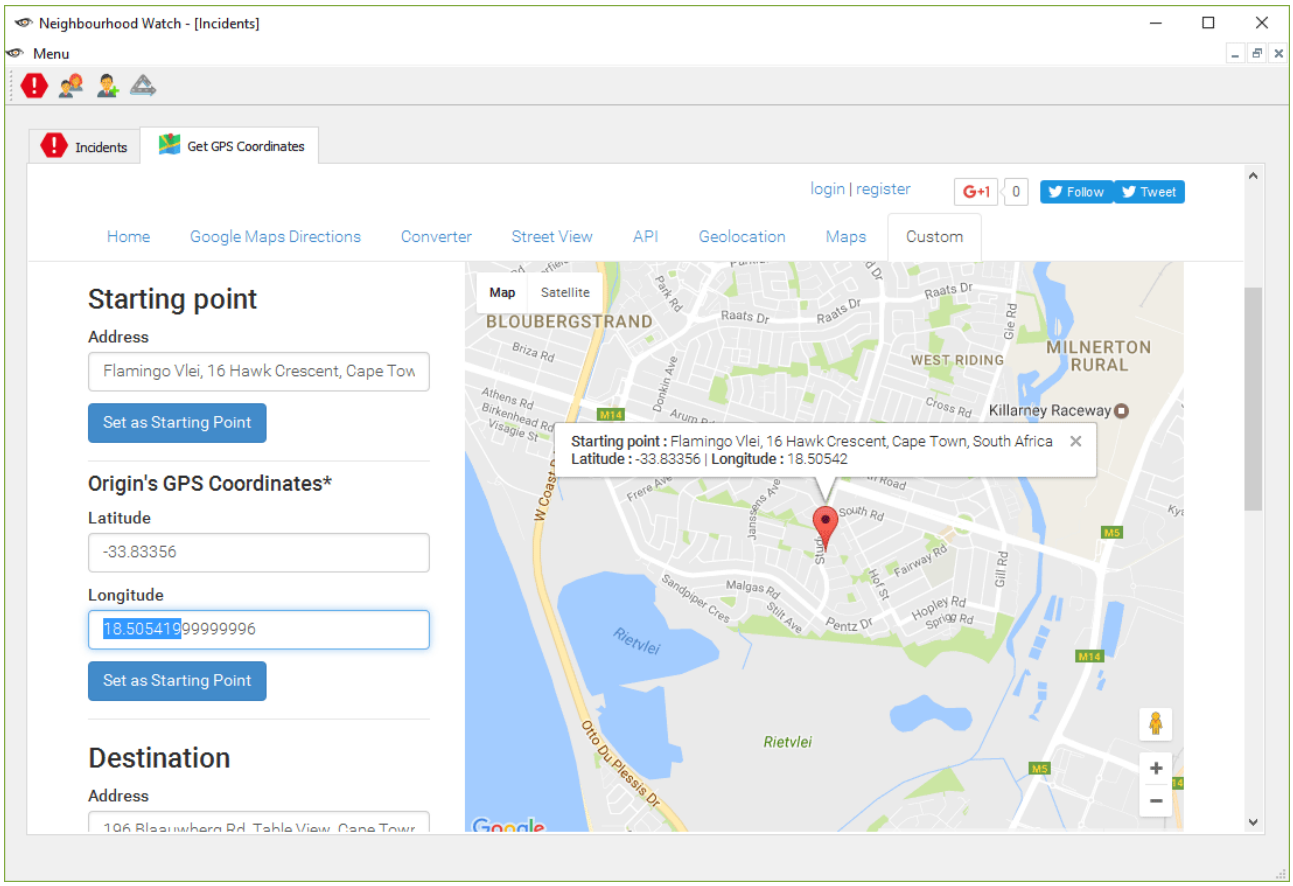
3. iv. a. Add new incident

All the incidents reported to date are listed in descending date order in the QTableView on the left. Adding a new incident is done using the 'Add New Incident' tab of the sub QTabWidget.

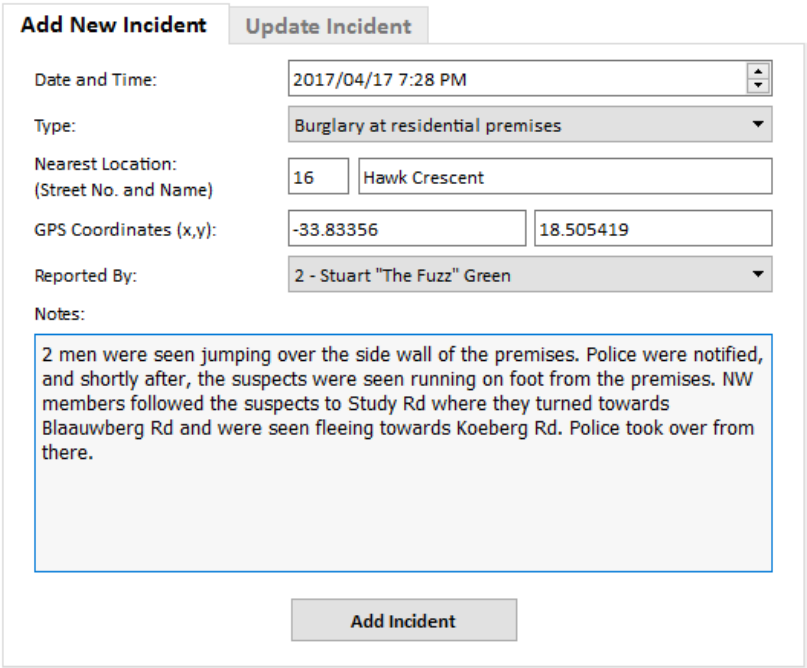
The two QComboBoxes are populated with values from the database, and the QDateTimeEdit has the current date and time set by default.



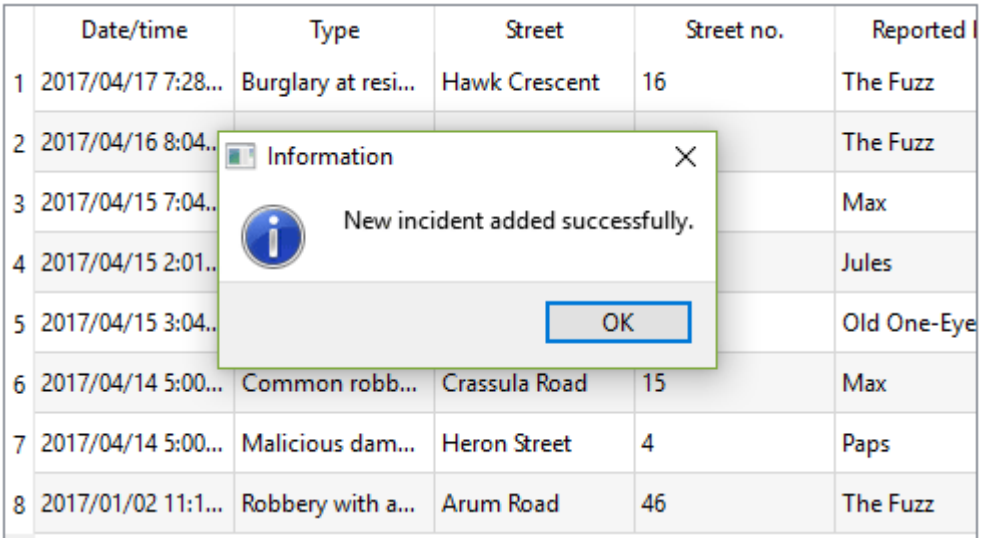
An incident location is recorded with the street address as well as GPS coordinates. I've implemented a QWebView widget in the 'Get GPS Coordinates' tab of the window's main QTabWidget which loads a website that uses Google Maps to provide GPS coordinates easily.



For example:

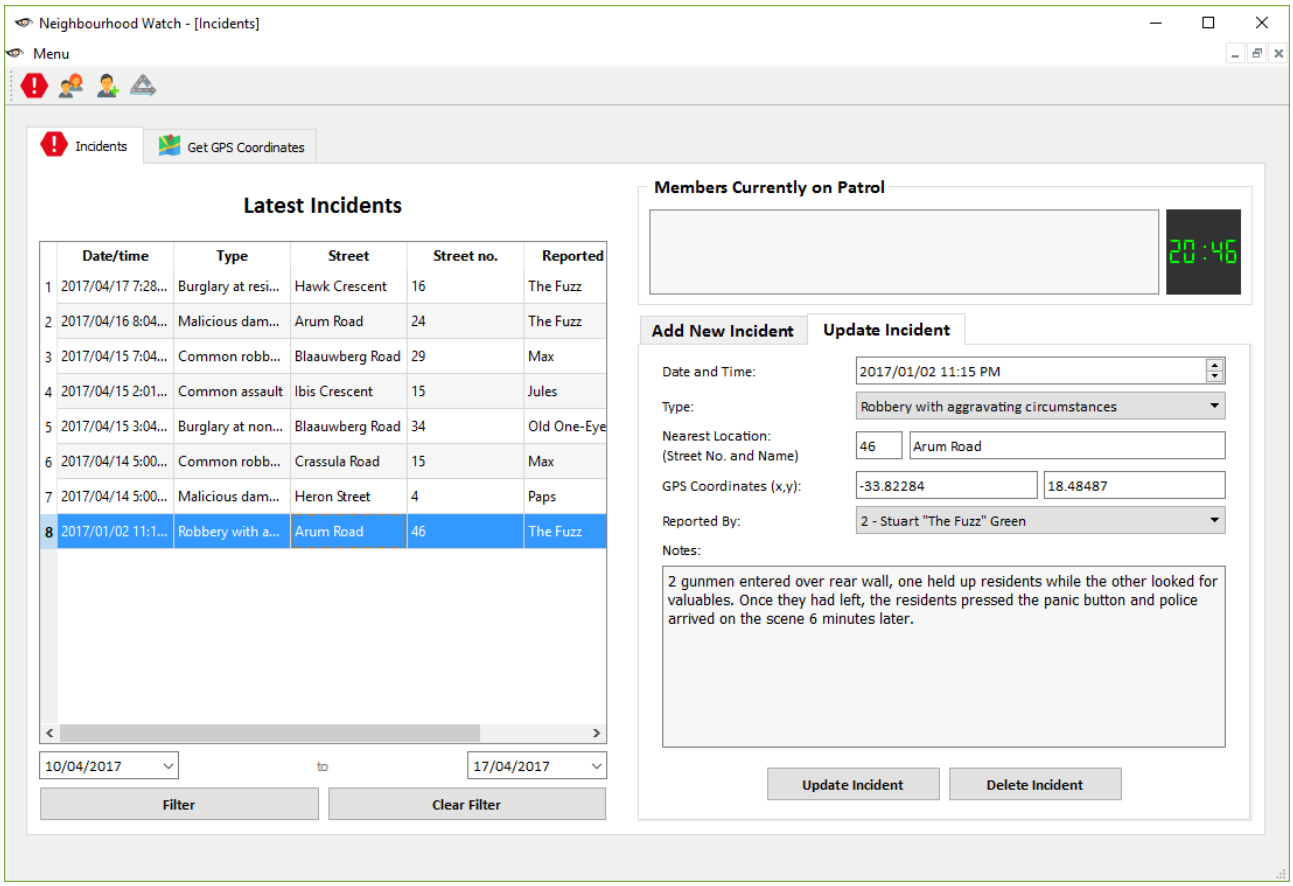


Clicking on 'Add Incident' will validate the fields and insert a new row into the database, and confirm the insertion with a QMessageBox.

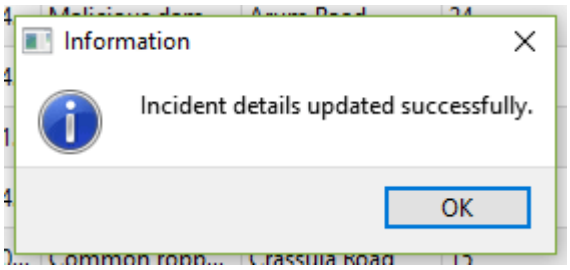


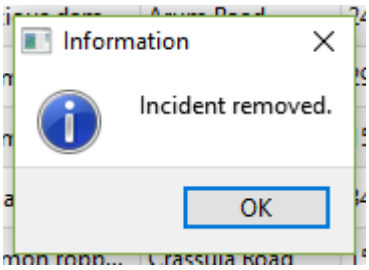
3. iv. b. Update or delete incident

Initially, the 'Update Incident' tab in the sub QTabWidget is disabled. When selecting an existing incident from the incidents QTableView, the tab is enabled, focused and populated with the selected incident's details (see following page).



The details can be modified and when 'Update Incident' is clicked, the same validation is done as the 'Add New Incident' tab and the row is updated in the database, with a QMessageBox confirming the update. Similarly, the user can click 'Delete Incident' and have the row removed from the database after confirming the action.



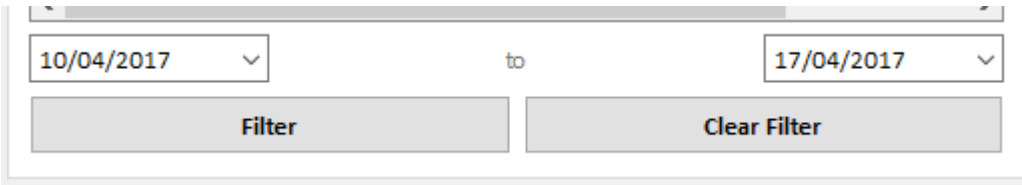


In both cases, the QTableView is also refreshed at this point to show the latest changes. The 'Update Incident' tab is also disabled since an incident is no longer selected. The 'Add New Incident' tab is focused.

3. iv. c. Filter by date range

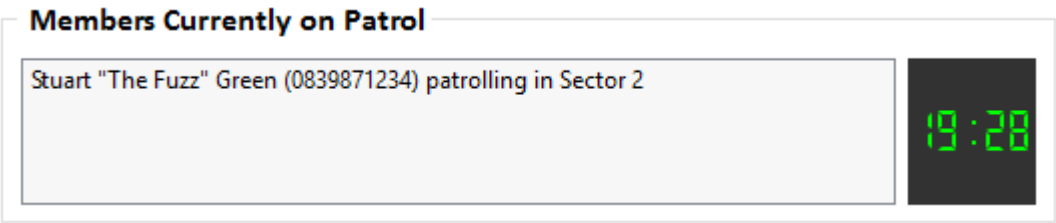
A user can filter the incidents by a date range chosen in the two QDateEdits. The default range is the last week. Clicking on the 'Filter' QPushButton will refresh the QTableView with the relevant results in date descending order.

Clicking on 'Clear Filter' will reload the original table with all the incidents listed.



3. iv. d. See currently patrolling members

This part of the application shows who is currently patrolling and where. The QListWidget is populated with concatenated information from the database based on the current time. This list is updated every minute automatically.



I also display the current time using QTime and the QLCDNumber widget which updates every second. Both features use QTimer to update themselves automatically.

4. Database manipulation

I will break up this section into the following groups and show the data management related code for each:

i. **Member management**
   a. Adding a member
   b. Adding a member photo
   c. Populating comboboxes
   d. Populating the members table view
   e. Updating a member

ii. **Patrol time slot management**
   a. Adding a patrol time slot
   b. Removing a patrol time slot

iii. **Incident management**
   a. Populating the incidents table
   b. Adding a new incident
   c. Updating an incident
   d. Deleting an incident
   e. Filter incidents

4. i. a. Adding a member

This function grabs the values from the UI elements, validates them, and prepares a query to execute to insert the information into the database.

```
def addNewMember(self):
    # Get the values from the GUI elements
    name = self.ui.editNewName.text()
    surname = self.ui.editNewSurname.text()
    callsign = self.ui.editNewCallsign.text()
    contact_number = self.ui.editNewContactNo.text()
    address = self.ui.editNewAddress.text()
    date_joined = self.ui.dateNewDateJoined.date()
    photo = self.ui.labelAddMemberPhoto.filename
    id_number = self.ui.editNewIdNo.text()
    status = 0
    if (self.ui.checkNewStatus.isChecked()):
        status = 1

    # Validation
    validation_message = ""
    if name == "":
        validation_message += "Name field cannot be empty.\n"
    if surname == "":
        validation_message += "Surname field cannot be empty.\n"
```

```
    if callsign == "":
        validation_message += "Callsign field cannot be empty.\n"
    if contact_number == "":
        validation_message += "Contact number field cannot be empty.\n"
    if id_number == "":
        validation_message += "ID number field cannot be empty.\n"
    if name.isdigit():
        validation_message += "Please enter a proper name.\n"
    if surname.isdigit():
        validation_message += "Please enter a proper surname.\n"
    if not str(contact_number).isdigit():
        validation_message += "Please enter a proper contact number.\n"
    if (not str(id_number).isdigit()) or (not len(str(id_number)) == 13):
        validation_message += "Please enter a proper ID number.\n"
    if (not len(str(contact_number)) < 11):
        validation_message += "Contact number is too long.\n"
    if not (validation_message == ""):
        showMessageBox(QMessageBox.Warning, "There are some problems with the
data.", validation_message, "Oops!")
        return False

    # Prepare sql query for insert
    query = QtSql.QSqlQuery()
    query.prepare("INSERT INTO member "
        "(name, surname, callsign, contact_number, address, date_joined, photo,
id_number, status) "
        "VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)")
    query.addBindValue(name)
    query.addBindValue(surname)
    query.addBindValue(callsign)
    query.addBindValue(contact_number)
    query.addBindValue(address)
    query.addBindValue(date_joined)
    query.addBindValue(photo)
    query.addBindValue(id_number)
    query.addBindValue(status)

    # Insert new row into member table of db
    if query.exec_():
        print ("Row added successfully")
    else:
        print (query.lastError().text())
        return False

    # Show confirmation message
    showMessageBox(QMessageBox.Information, "New member added successfully.", "",
"Information")

    # Update member table view
    self.populateMemberTableView()

    # Reset fields to empty values
    self.ui.editNewName.setText("")
    self.ui.editNewName.setFocus()
```

```
    self.ui.editNewSurname.setText("")
    self.ui.editNewCallsign.setText("")
    self.ui.editNewContactNo.setText("")
    self.ui.editNewAddress.setText("")
    self.ui.editNewIdNo.setText("")

    # Remove the pixmap from the photo label
    pixmap = QtGui.QPixmap("")
    self.ui.labelAddMemberPhoto.setPixmap(pixmap)
```

4. i. b. Adding a member photo

To load a photo into the GUI I call `getPhotoPath()` from `addMemberPhoto()` to open a QFileDialog to get the file path. Then I create a pixmap and assign it to the QLabel. I also set a 'filename' attribute on it to use later when saving the file path to the database.

```
def getPhotoPath(self):
    # Open file dialog to get photo file
    file_dialog = QtGui.QFileDialog()
    file_dialog.setWindowTitle('Open photo file')
    file_dialog.setNameFilter('Images (*.png *.gif *.jpg)')
    if file_dialog.exec_():
        filename = file_dialog.selectedFiles()[0]
        return filename
    else:
        return False

def addMemberPhoto(self):
    # Call function to get the file path
    filename = self.getPhotoPath()

    # Set the label's pixmap to the file path
    if str(type(filename)) == "<class 'str'>":
        self.ui.labelAddMemberPhoto.setPixmap(QtGui.QPixmap(filename))
        self.ui.labelAddMemberPhoto.setScaledContents(True)
        self.ui.labelAddMemberPhoto.filename = filename
```

4. i. c. Populating comboboxes

When the application loads up I call these functions to populate the 'Patrol Time Slot' and 'Area' QComboBoxes of the 'Patrols' tab on the 'Manage Members' window (see bottom page 10).

```
def populatePatrolTimeSlotsCombo(self):
    # Get the selected week day
```

```
    week_day = str(self.ui.comboAddPatrolWeekDay.currentText())

    # Clear the time slots in the combo in preparation for new list
    self.ui.comboAddPatrolTimeSlot.clear()

    # Prepare query to fetch time slots based on week day
    query = QtSql.QSqlQuery()
    query.prepare("SELECT time_start, time_end FROM patrol_time WHERE week_day
= ?")
    query.addBindValue(week_day)

    # Execute the query and build the string to append to the combobox
    query.exec_()
    while query.next():
        time_start = str(query.value(0).toString("hh:mm"))
        time_end = str(query.value(1).toString("hh:mm"))
        time_slot = time_start + " - " + time_end
        self.ui.comboAddPatrolTimeSlot.addItem(time_slot)

def populatePatrolAreaCombo(self):
    # Prepare and execute query to get descriptions from patrol_area
    query = QtSql.QSqlQuery()
    query.prepare("SELECT description FROM patrol_area")
    query.exec_()
    while query.next():
        area = query.value(0)
        # Append to combo box
        self.ui.comboAddPatrolArea.addItem(area)
```

4. i. d. Populating the members table view

When the application starts up, I populate the members QTableView in the 'Manage Members' window.

```
def populateMemberTableView(self):
    # Set the model of the table to the member table from the db
    self.model = QtSql.QSqlTableModel(self)
    self.model.setTable("member")
    self.model.setEditStrategy(QtSql.QSqlTableModel.OnManualSubmit)
    self.model.select()

    # Set custom names for headers
    self.model.setHeaderData(1, QtCore.Qt.Horizontal, "Name")
    self.model.setHeaderData(2, QtCore.Qt.Horizontal, "Surname")
    self.model.setHeaderData(3, QtCore.Qt.Horizontal, "Callsign")
    self.model.setHeaderData(4, QtCore.Qt.Horizontal, "Contact no.")

    # Hide columns I don't want to show in the table view
    self.ui.tableMembers.setModel(self.model)
    self.ui.tableMembers.hideColumn(0)
```

```
    self.ui.tableMembers.hideColumn(5)
    self.ui.tableMembers.hideColumn(6)
    self.ui.tableMembers.hideColumn(7)
    self.ui.tableMembers.hideColumn(8)
    self.ui.tableMembers.hideColumn(9)
```

4. i. e. Updating a member

When selecting a row in the member QTableView the `populateUpdateMemberForms()` function is called which enables the QTabWidget and populates all the fields with data from the database. It also calls the `populateLinkedPatrolsTableView(member_id)` function which populates the 'Current Patrols' table in the 'Patrols' tab for that member as well (see pages 10-11).

```
def populateUpdateMemberForms(self):
    # Get the member_id of the selected row
    selected_row = self.ui.tableMembers.selectionModel().selectedRows()[0]
    member_id = self.ui.tableMembers.model().data(selected_row)

    # Enable tab widget
    self.ui.tabWidgetMembers.setEnabled(True)

    # Get the row of data for the member
    query = QtSql.QSqlQuery()
    query.prepare("SELECT * FROM member WHERE member_id = ?")
    query.addBindValue(member_id)
    query.exec_()
    while(query.next()):
        # Populate all the GUI fields with the data
        self.ui.editManageName.setText(query.value(1))
        self.ui.editManageSurname.setText(query.value(2))
        self.ui.editManageCallsign.setText(query.value(3))
        self.ui.editManageIdNo.setText(str(query.value(8)))
        self.ui.editManageContactNo.setText(str(query.value(4)))
        self.ui.editManageAddress.setText(query.value(5))
        self.ui.dateManageDateJoined.setDate(query.value(6))
        if (query.value(9) == 1):
            self.ui.checkManageStatus.setChecked(True)
        else:
            self.ui.checkManageStatus.setChecked(False)

        # Use the photo file path to load a pixmap into a label
        file_path = query.value(7)

        if str(type(file_path)) == "<class 'str'>":
            pixmap = QtGui.QPixmap(file_path)
            self.ui.labelChangeMemberPhoto.setPixmap(pixmap)
            self.ui.labelChangeMemberPhoto.setScaledContents(True)
        else:
            pixmap = QtGui.QPixmap("")
            self.ui.labelChangeMemberPhoto.setPixmap(pixmap)
```

```
        # Call function to populate Patrols table view
        self.populateLinkedPatrolsTableView(member_id)

def populateLinkedPatrolsTableView(self, member_id):
    # Set the model of the table view to the linked_patrols view in the db
    self.model = QtSql.QSqlTableModel(self)
    self.model.setTable("linked_patrols")
    self.model.setFilter("member_id like '" + str(member_id) + "'")
    self.model.setEditStrategy(QtSql.QSqlTableModel.OnManualSubmit)
    self.model.select()

    # Change column headers to custom names
    self.model.setHeaderData(3, QtCore.Qt.Horizontal, "Week day")
    self.model.setHeaderData(4, QtCore.Qt.Horizontal, "Start time")
    self.model.setHeaderData(5, QtCore.Qt.Horizontal, "End time")
    self.model.setHeaderData(6, QtCore.Qt.Horizontal, "Patrol area")

    # Hide columns I don't want to see on the table view
    self.ui.tableLinkedPatrols.setModel(self.model)
    self.ui.tableLinkedPatrols.hideColumn(0)
    self.ui.tableLinkedPatrols.hideColumn(1)
    self.ui.tableLinkedPatrols.hideColumn(2)
    self.ui.tableLinkedPatrols.resizeColumnsToContents()
```

A user can now edit the member's personal details and save the changes by clicking the 'Save Changes' button which calls the `updateMemberDetails()` function.

```
def updateMemberDetails(self):
    # Get the member_id from the selected table row
    selected_member_row = self.ui.tableMembers.selectionModel().selectedRows()[0]
    member_id = self.ui.tableMembers.model().data(selected_member_row)

    # Grab the values from all the GUI fields
    name = self.ui.editManageName.text()
    surname = self.ui.editManageSurname.text()
    callsign = self.ui.editManageCallsign.text()
    id_number = self.ui.editManageIdNo.text()
    contact_number = self.ui.editManageContactNo.text()
    address = self.ui.editManageAddress.text()
    date_joined = self.ui.dateManageDateJoined.date()
    status = 0
    if (self.ui.checkManageStatus.isChecked()):
        status = 1
    photo = self.ui.labelChangeMemberPhoto.filename

    # Validation
    validation_message = ""
    if name == "":
        validation_message += "Name field cannot be empty.\n"
    if surname == "":
```

```
        validation_message += "Surname field cannot be empty.\n"
    if callsign == "":
        validation_message += "Callsign field cannot be empty.\n"
    if contact_number == "":
        validation_message += "Contact number field cannot be empty.\n"
    if id_number == "":
        validation_message += "ID number field cannot be empty.\n"
    if name.isdigit():
        validation_message += "Please enter a proper name.\n"
    if surname.isdigit():
        validation_message += "Please enter a proper surname.\n"
    if not str(contact_number).isdigit():
        validation_message += "Please enter a proper contact number.\n"
    if (not str(id_number).isdigit()) or (not len(str(id_number)) == 13):
        validation_message += "Please enter a proper ID number.\n"
    if (not len(str(contact_number)) < 11):
        validation_message += "Contact number is too long.\n"
    if not (validation_message == ""):
        showMessageBox(QMessageBox.Warning, "There are some problems with the
data.", validation_message, "Oops!")
        return False

    # Prepare query to update member details in db
    query = QtSql.QSqlQuery()
    query.prepare("UPDATE member SET "
        "name = ?, surname = ?, callsign = ?, contact_number = ?, address = ?,
date_joined = ?, photo = ?, "
        "id_number = ?, status = ? WHERE member_id = ?")
    query.addBindValue(name)
    query.addBindValue(surname)
    query.addBindValue(callsign)
    query.addBindValue(contact_number)
    query.addBindValue(address)
    query.addBindValue(date_joined)
    query.addBindValue(photo)
    query.addBindValue(id_number)
    query.addBindValue(status)
    query.addBindValue(member_id)

    # Execute query
    if query.exec_():
        print ("Row added successfully")
        self.populateMemberTableView()
        showMessageBox(QMessageBox.Information, "Member details updated
successfully.", "", "Information")
    else:
        print (query.lastError().text())
```

On the 'Patrols' tab of a member, the user can add a new patrol to the member's schedule by selecting options in the QComboBoxes and hitting 'Add Patrol'. The `addPatrolLink()` function is called which gets the selected values from the GUI elements, and first does a check to see whether the member is already patrolling that time. If not, it inserts the new patrol for that member into the database, and then calls

functions to update the member's 'Current Patrols' QTableView and the 'Members Currently on Patrol' QListView on the 'Incidents' window (see page 13).

```
def addPatrolLink(self):
    # Get the member_id from the selected table row
    selected_member_row = self.ui.tableMembers.selectionModel().selectedRows()[0]
    member_id = self.ui.tableMembers.model().data(selected_member_row)

    # Grab the selected values from the comboboxes
    week_day = str(self.ui.comboAddPatrolWeekDay.currentText())
    time_slot = str(self.ui.comboAddPatrolTimeSlot.currentText())
    time_start, x, time_end = time_slot.split()
    area = str(self.ui.comboAddPatrolArea.currentText())

    # Prepare and execute the sql query to get the time_id of the selected time
slot
    query = QtSql.QSqlQuery()
    query.prepare("SELECT time_id FROM patrol_time WHERE week_day = ? AND
time_start = ? AND time_end = ?")
    query.addBindValue(week_day)
    query.addBindValue(time_start)
    query.addBindValue(time_end)
    if not query.exec_():
        print(query.lastError().text())
    while query.next():
        time_id = query.value(0)

    # Prepare and execute the sql query to get the area_id of the selected area
    query.prepare("SELECT area_id FROM patrol_area WHERE description = ?")
    query.addBindValue(area)
    if not query.exec_():
        print(query.lastError().text())
    while query.next():
        area_id = query.value(0)

    # Prepare and execute the sql queries to check whether the member is already
patrolling at those times and if not, insert the row
    query.prepare("SELECT * FROM patrol WHERE member_id = ? AND time_id = ? AND
area_id = ?")
    query.addBindValue(member_id)
    query.addBindValue(time_id)
    query.addBindValue(area_id)
    if not query.exec_():
        print(query.lastError().text())
    if query.size() > 0:
        # If a row exists, the member is already patrolling that time
        showMessageBox(QMessageBox.Warning, "Member already linked to that
patrol.", "", "Oops!")
    else:
        # If there is no row, insert the new patrol for the member
        query.prepare("INSERT INTO patrol (member_id, time_id, area_id) VALUES
(?, ?, ?)")
        query.addBindValue(member_id)
```

```
        query.addBindValue(time_id)
        query.addBindValue(area_id)
        if not query.exec_():
            print(query.lastError().text())
        else:
            print("Row added successfully")

        # Refresh the member's patrol table view and the currently patrolling
list
        self.populateLinkedPatrolsTableView(member_id)
        self.populateCurrentlyPatrollingList()
        showMessageBox(QMessageBox.Information, "New patrol linked
successfully.", "", "Information")
```

A patrol can also be removed from the member's schedule by calling the `removePatrolLink()` function. The function checks that a table row is selected first, and then asks for confirmation before deleting the patrol from the member's schedule.

```
def removePatrolLink(self):
    # Check that a table row has been selected
    if (self.ui.tableLinkedPatrols.currentIndex().row() == -1):
        msg = QMessageBox()
        msg.setIcon(QMessageBox.Information)
        msg.setText("You need to select a linked patrol first.")
        msg.setWindowTitle("Information")
        msg.setStandardButtons(QMessageBox.Ok)
        msg.exec_()
        return False
    # Ask for confirmation and delete the db row
    else:
        msg = QMessageBox()
        msg.setIcon(QMessageBox.Question)
        msg.setText("Are you sure you want to remove\nthis linked patrol?")
        msg.setWindowTitle("Confirm")
        msg.setStandardButtons(QMessageBox.Yes | QMessageBox.No)
        ret = msg.exec_()
        if (ret == QMessageBox.Yes):
            # Get the selected row
            selected_row =
self.ui.tableLinkedPatrols.selectionModel().selectedRows()[0]

            # Get the required values from the selected row
            member_id =
self.ui.tableLinkedPatrols.model().index(selected_row.row(), 0).data()
            time_id =
self.ui.tableLinkedPatrols.model().index(selected_row.row(), 1).data()
            area_id =
self.ui.tableLinkedPatrols.model().index(selected_row.row(), 2).data()

            # Prepare and execute query to delete matching patrol from the db
            query = QtSql.QSqlQuery()
```

```
            query.prepare("DELETE FROM patrol WHERE member_id = ? AND time_id = ?
AND area_id = ?")
            query.addBindValue(member_id)
            query.addBindValue(time_id)
            query.addBindValue(area_id)
            if not query.exec_():
                print(query.lastError().text())
                return False

            # Reload the member's patrol table view
            self.populateLinkedPatrolsTableView(member_id)
            showMessageBox(QMessageBox.Information, "Linked patrol time
removed.", "", "Information")
        else:
            return False
```

4. ii. a. Adding a patrol time slot

On the 'Manage Patrol Time Slots' window (see page 7) you can make new combinations of hours and minutes (in 30 minute blocks) on week days to create a new patrol time slot. Clicking on the 'Add Time Slot' QPushButton calls the `addNewTimeSlot()` function. The functions takes the concatenates the hours and minutes and compares them, together with week day, to the existing slots in the database to see if such a combination already exists. If it doesn't, it gets added to the database and the 'Available Patrol Slots' QTableView is reloaded.

```
def addNewTimeSlot(self):
    # Get the values from the GUI elements
    week_day = str(self.ui.comboWeekDay.currentText())
    start_hour = str(self.ui.comboStartHour.currentText())
    start_min = str(self.ui.comboStartMin.currentText())
    end_hour = str(self.ui.comboEndHour.currentText())
    end_min = str(self.ui.comboEndMin.currentText())

    # Concatenate hours and minutes to get strings to compare to in the db
    time_start = str(start_hour + ":" + start_min)
    time_end = str(end_hour + ":" + end_min)

    # Validation
    if start_hour == end_hour:
        validation_message = "A patrol must be at least one\nhour in length."
    if not (validation_message == ""):
        showMessageBox(QMessageBox.Warning, validation_message, "", "Oops!")
        return False

    # Check if the record already exists
    query = QtSql.QSqlQuery()
    query.prepare("SELECT (1) FROM patrol_time WHERE week_day = ? AND time_start
= ? AND time_end = ?")
    query.addBindValue(week_day)
```

```
    query.addBindValue(time_start)
    query.addBindValue(time_end)
    query.exec_()
    if query.size() > 0:
        showMessageBox(QMessageBox.Warning, "That patrol time slot already
exists!\nHint: You can order the table by day or time to find it.", "", "Oops!")
        return False

    # Insert new row into patrol_time table of db
    query.prepare("INSERT INTO patrol_time "
        "(week_day, time_start, time_end) "
        "VALUES (?, ?, ?)")
    query.addBindValue(week_day)
    query.addBindValue(time_start)
    query.addBindValue(time_end)
    if query.exec_():
        print ("Row added successfully")
    else:
        print (query.lastError().text())

    # Update patrol time table view
    self.populatePatrolTimesTableView()

    showMessageBox(QMessageBox.Information, "New patrol time slot added
successfully.", "", "Information")
```

4. ii. b. Removing a patrol time slot

The `removeTimeSlot()` function checks to see if a table row is selected in the 'Available Patrol Slots' QTableView and, after getting confirmation, removes the row from the table. The model linked to the table view is then submitted to save the deletion to the database.

```
def removeTimeSlot(self):
    # Check if a table row is selected
    if (self.ui.tablePatrolTimes.currentIndex().row() == -1):
        msg = QMessageBox()
        msg.setIcon(QMessageBox.Information)
        msg.setText("You need to select a patrol time slot first.")
        msg.setWindowTitle("Information")
        msg.setStandardButtons(QMessageBox.Ok)
        msg.exec_()
        return False
    # Ask for confirmation of deletion
    else:
        msg = QMessageBox()
        msg.setIcon(QMessageBox.Question)
        msg.setText("Are you sure you want to remove\nthis time slot?")
        msg.setWindowTitle("Confirm")
        msg.setStandardButtons(QMessageBox.Yes | QMessageBox.No)
```

```
        ret = msg.exec_()
        if (ret == QMessageBox.Yes):
            # Remove the row from the table and submit the changes
            self.model.removeRow(self.ui.tablePatrolTimes.currentIndex().row())
            self.model.submitAll()
            # Check if there is an error from the db
            if self.model.lastError():
                showMessageBox(QMessageBox.Information, "A member patrol is
linked to that time slot.\nIt cannot be removed until the patrol is unlinked.",
"", "Information")
            else:
                print ("Row removed.")
                showMessageBox(QMessageBox.Information, "Patrol time slot
removed.", "", "Information")
        else:
            return False
```

4. iii. a. Populating the incidents table

When the application starts, I call the `populateIncidentTableView()` function to populate the 'Latest Incidents' QTableView (see page 13). I use the QSqlRelationalTableModel class to get the columns from the tables that are linked to the `incident` table, and also sort the results by the 'date_time' column in descending order.

```
def populateIncidentTableView(self):
    self.model = QtSql.QSqlRelationalTableModel(self)
    self.model.setTable("incident")
    self.model.setEditStrategy(QtSql.QSqlTableModel.OnManualSubmit)

    # Sort results by date_time descending
    self.model.setSort(1, QtCore.Qt.SortOrder(1))

    # Get columns from related tables
    self.model.setRelation(2, QtSql.QSqlRelation("incident_type", "type_id",
"description"))
    self.model.setRelation(7, QtSql.QSqlRelation("member", "member_id",
"callsign"))

    self.model.select()

    # Change display names of column headers
    self.model.setHeaderData(1, QtCore.Qt.Horizontal, "Date/time")
    self.model.setHeaderData(2, QtCore.Qt.Horizontal, "Type")
    self.model.setHeaderData(3, QtCore.Qt.Horizontal, "Street")
    self.model.setHeaderData(4, QtCore.Qt.Horizontal, "Street no.")
    self.model.setHeaderData(7, QtCore.Qt.Horizontal, "Reported by")

    # Hide columns I don't want to see on the table view
    self.ui.tableIncidents.setModel(self.model)
```

```
    self.ui.tableIncidents.hideColumn(0)
    self.ui.tableIncidents.hideColumn(5)
    self.ui.tableIncidents.hideColumn(6)
    self.ui.tableIncidents.hideColumn(8)
```

4. iii. b. Adding a new incident

When the application starts, I populate the 'Type' and 'Reported By' QComboBoxes in the 'Add New/Update Incident' sub QTabWidget in the main 'Incidents' tab (see page 14). The list of incident type descriptions is stored in the `incident_type` database table, and the list of members combobox contains information from the `member` table in the database. The `populateIncidentTypesCombo()` and `populateMembersCombo()` functions retrieve and show this data.

```
def populateIncidentTypesCombo(self):
    # Prepare and execute the query to get incident_type descriptions
    query = QtSql.QSqlQuery()
    query.prepare("SELECT description FROM incident_type")
    query.exec_()
    while query.next():
        # Append each to the combobox
        incident_type = query.value(0)
        self.ui.comboNewIncidentType.addItem(incident_type)
        self.ui.comboUpdateIncidentType.addItem(incident_type)

def populateMembersCombo(self):
    # Prepare and execure the the query to get member details
    query = QtSql.QSqlQuery()
    query.prepare("SELECT member_id, name, surname, callsign FROM member")
    query.exec_()
    while query.next():
        # For each member, create a string containing member_id, name, callsign,
surname, contact_number
        member_id = str(query.value(0))
        name = query.value(1)
        surname = query.value(2)
        callsign = query.value(3)
        full_name = member_id + " - " + name + " \"" + callsign + "\" " + surname
        # Add string to 'add new' tab and 'update' tab comboboxes
        self.ui.comboNewIncidentReportedBy.addItem(full_name)
        self.ui.comboUpdateIncidentReportedBy.addItem(full_name)
```

To add an incident, the user can fill in the form in the 'Add New Incident' tab. The 'Date and Time' QDateTimeEdit is automatically populated with the current time and date. Once the form is filled in, clicking 'Add Incident' will call the `addNewIncident()` function and validate the data

```
def addNewIncident(self):
    # Get field values from the GUI elements
    date_time = self.ui.dateNewIncidentDateTime.dateTime()
    location_street = self.ui.editNewIncidentStreetName.text()
    location_number = self.ui.editNewIncidentStreetNo.text()
    gps_x = self.ui.editNewIncidentGpsX.text()
    gps_y = self.ui.editNewIncidentGpsY.text()
    notes = self.ui.textNewIncidentNotes.toPlainText()

    # Get type id of incident by matching type description to db
    type_description = self.ui.comboNewIncidentType.currentText()
    query = QtSql.QSqlQuery()
    query.prepare("SELECT type_id FROM incident_type WHERE description = ?")
    query.addBindValue(type_description)
    query.exec_()
    while query.next():
        type_id = query.value(0)

    # Get the member id from combo box text
    member_full_name = self.ui.comboNewIncidentReportedBy.currentText()
    reported_by_member_id = member_full_name.split()[0]

    # Validation
    validation_message = ""
    if location_street == "":
        validation_message += "Street name cannot be empty.\n"
    if location_number == "":
        validation_message += "Street number cannot be empty.\n"
    if gps_x == "" or gps_y == "":
        validation_message += "GPS coordinates cannot be empty.\n"
    if not (validation_message == ""):
        showMessageBox(QMessageBox.Warning, "There are some problems with the
data.", validation_message, "Oops!")
        return False

    # Prepare query for inserting new incident into db
    query.prepare("INSERT INTO incident "
        "(date_time, type_id, location_street, location_number, gps_x, gps_y,
reported_by_member_id, notes) VALUES "
        "(?, ?, ?, ?, ?, ?, ?, ?)")
    query.addBindValue(date_time)
    query.addBindValue(type_id)
    query.addBindValue(location_street)
    query.addBindValue(location_number)
    query.addBindValue(gps_x)
    query.addBindValue(gps_y)
    query.addBindValue(reported_by_member_id)
    query.addBindValue(notes)
    # Execute query
    if not query.exec_():
        print(query.lastError().text())
    else:
        print("Row added successfully")
        # Refresh table view with latest incidents
```

```
        self.populateIncidentTableView()
        showMessageBox(QMessageBox.Information, "New incident added
successfully.", "", "Information")

    # Reset fields
    self.ui.editNewIncidentStreetName.setText("")
    self.ui.editNewIncidentStreetNo.setText("")
    self.ui.editNewIncidentGpsX.setText("")
    self.ui.editNewIncidentGpsY.setText("")
    self.ui.textNewIncidentNotes.clear()
```

4. iii. c. Updating an incident

Clicking on a row of the 'Latest Incidents' table calls the `populateUpdateIncidentForm()` function which gets the 'incident_id' of the selected row and populates the 'Update Incident' tab's form with the values (see page 16). It also sets the indices of the QComboBoxes to the correct selection that matches the information in the database.

```
def populateUpdateIncidentForm(self):
    # Get the incident_id of the selected row
    selected_row = self.ui.tableIncidents.selectionModel().selectedRows()[0]
    incident_id = self.ui.tableIncidents.model().data(selected_row)

    # Enable and focus the update tab
    self.ui.tabWidgetIncidents.setTabEnabled(1, True)
    self.ui.tabWidgetIncidents.setCurrentIndex(1)

    # Get the incident data
    query = QtSql.QSqlQuery()
    query.prepare("SELECT * FROM incident WHERE incident_id = ?")
    query.addBindValue(incident_id)
    query.exec_()
    while query.next():
        self.ui.dateUpdateIncidentDateTime.setDateTime(query.value(1))
        type_id = query.value(2)
        self.ui.editUpdateIncidentStreetNo.setText(str(query.value(4)))
        self.ui.editUpdateIncidentStreetName.setText(query.value(3))
        self.ui.editUpdateIncidentGpsX.setText(str(query.value(5)))
        self.ui.editUpdateIncidentGpsY.setText(str(query.value(6)))
        reported_by_member_id = query.value(7)
        self.ui.textUpdateIncidentNotes.clear()
        self.ui.textUpdateIncidentNotes.insertPlainText(query.value(8))

    # Get the incident description from type_id
    query.prepare("SELECT description FROM incident_type WHERE type_id = ?")
    query.addBindValue(type_id)
    query.exec_()
    while query.next():
        type_description = query.value(0)
```

```
    # Set the 'incident type' combo box to the correct index
    incident_type_index =
self.ui.comboUpdateIncidentType.findText(type_description,
QtCore.Qt.MatchFixedString)
    self.ui.comboUpdateIncidentType.setCurrentIndex(incident_type_index)

    # Set the 'reported by member' combo box to the correct index
    member_index =
self.ui.comboUpdateIncidentReportedBy.findText(str(reported_by_member_id) + " ",
QtCore.Qt.MatchStartsWith)
    self.ui.comboUpdateIncidentReportedBy.setCurrentIndex(member_index)
```

The user can now update the details of the incident and clicking on 'Update Incident' calls the `updateIncident()` function which is similar to the `addNewIncident()` function, except that the SQL is an update query.

```
def updateIncident(self):
    # Get incident id from selected row
    selected_incident_row =
self.ui.tableIncidents.selectionModel().selectedRows()[0]
    incident_id = self.ui.tableIncidents.model().data(selected_incident_row)

    # Get field values from GUI elements
    date_time = self.ui.dateUpdateIncidentDateTime.dateTime()
    location_street = self.ui.editUpdateIncidentStreetName.text()
    location_number = self.ui.editUpdateIncidentStreetNo.text()
    gps_x = self.ui.editUpdateIncidentGpsX.text()
    gps_y = self.ui.editUpdateIncidentGpsY.text()
    notes = self.ui.textUpdateIncidentNotes.toPlainText()

    # Get type id of incident
    type_description = self.ui.comboUpdateIncidentType.currentText()
    query = QtSql.QSqlQuery()
    query.prepare("SELECT type_id FROM incident_type WHERE description = ?")
    query.addBindValue(type_description)
    query.exec_()
    while query.next():
        type_id = query.value(0)

    # Get member id from combo box text
    member_full_name = self.ui.comboUpdateIncidentReportedBy.currentText()
    reported_by_member_id = member_full_name.split()[0]

    # Validation
    validation_message = ""
    if location_street == "":
        validation_message += "Street name cannot be empty.\n"
    if location_number == "":
        validation_message += "Street number cannot be empty.\n"
    if gps_x == "" or gps_y == "":
        validation_message += "GPS coordinates cannot be empty.\n"
```

```
    if not (validation_message == ""):
        showMessageBox(QMessageBox.Warning, "There are some problems with the
data.", validation_message, "Oops!")
        return False

    # Update incident in db with new values
    query.prepare("UPDATE incident SET "
        "date_time = ?, type_id = ?, location_street = ?, location_number = ?,
gps_x = ?, gps_y = ?, "
        "reported_by_member_id = ?, notes = ? WHERE incident_id = ?")
    query.addBindValue(date_time)
    query.addBindValue(type_id)
    query.addBindValue(location_street)
    query.addBindValue(location_number)
    query.addBindValue(gps_x)
    query.addBindValue(gps_y)
    query.addBindValue(reported_by_member_id)
    query.addBindValue(notes)
    query.addBindValue(incident_id)
    if not query.exec_():
        print(query.lastError().text())
    else:
        print("Row updated successfully")
        self.populateIncidentTableView()
        showMessageBox(QMessageBox.Information, "Incident details updated
successfully.", "", "Information")
```

4. iii. d. Deleting an incident

On the same 'Update Incident' tab, the user can delete an incident by clicking the 'Delete Incident' QPushButton. The `deleteIncident()` function gets the 'incident_id' of the currently selected table row and removes the row from the database, before reloading the 'Latest Incidents' table view and disabling the 'Update Incident' tab (since no incident is selected anymore).

```
def deleteIncident(self):
    # Ask for confirmation of deletion
    msg = QMessageBox()
    msg.setIcon(QMessageBox.Question)
    msg.setText("Are you sure you want to delete this incident?")
    msg.setWindowTitle("Confirm")
    msg.setStandardButtons(QMessageBox.Yes | QMessageBox.No)
    ret = msg.exec_()
    if (ret == QMessageBox.Yes):
        # Get the currently selected incident_id
        selected_row = self.ui.tableIncidents.selectionModel().selectedRows()[0]
        incident_id = self.ui.tableIncidents.model().index(selected_row.row(),
0).data()

        # Delete from db
```

```
        query = QtSql.QSqlQuery()
        query.prepare("DELETE FROM incident WHERE incident_id = ?")
        query.addBindValue(incident_id)
        if not query.exec_():
            print(query.lastError().text())
            return False

        # Reload the incidents table
        self.populateIncidentTableView()

        # Disable the update tab because incident is not there anymore
        self.ui.tabWidgetIncidents.setCurrentIndex(0)
        self.ui.tabWidgetIncidents.setTabEnabled(1, False)

        showMessageBox(QMessageBox.Information, "Incident removed.", "",
"Information")
    else:
        return False
```

4. iii. e. Filter incidents

By default, the two QDateEdit widgets below the 'Latest Incidents' table are set to a range of the last week. When the 'Filter' button is clicked, the `filterIncidents()` function disables the 'Update Incident' tab since no incident is selected anymore, and validates that the 'To' date is after the 'From' date before setting a filter on the QSqlRelationalTableModel between the two selected dates.

```
def filterIncidents(self):
    # Disable the update tab because incident is not selected anymore
    self.ui.tabWidgetIncidents.setCurrentIndex(0)
    self.ui.tabWidgetIncidents.setTabEnabled(1, False)

    # Convoluted way to get date and time for sql statement :/
    from_date = self.ui.dateIncidentFilterFrom.date().toPyDate()
    to_date = self.ui.dateIncidentFilterTo.date().toPyDate()
    from_time = datetime.time(datetime(2000, 1, 1, 0, 0) - timedelta(seconds=1))
    to_time = datetime.time(datetime(2000, 1, 1, 0, 0) + timedelta(seconds=1))
    from_date_time = datetime.combine(from_date, from_time)
    to_date_time = datetime.combine(to_date, to_time)

    # Reload table with incidents matching provided date range
    if to_date < from_date:
        showMessageBox(QMessageBox.Warning, "'To' date cannot be before 'From'
date.", "", "Oops!")
        return False

    self.model = QtSql.QSqlRelationalTableModel(self)
    self.model.setTable("incident")
    self.model.setEditStrategy(QtSql.QSqlTableModel.OnManualSubmit)
```

```
    # Find the incidents that match the date range and set the sort order
    self.model.setFilter("date_time > '" + str(from_date) + "'AND date_time < '"
+ str(to_date) + "'")
    self.model.setSort(1, QtCore.Qt.SortOrder(1))

    # Get real values for display via relations
    self.model.setRelation(2, QtSql.QSqlRelation("incident_type", "type_id",
"description"))
    self.model.setRelation(7, QtSql.QSqlRelation("member", "member_id",
"callsign"))
    self.model.select()

    # Change display name of column headers
    self.model.setHeaderData(1, QtCore.Qt.Horizontal, "Date/time")
    self.model.setHeaderData(2, QtCore.Qt.Horizontal, "Type")
    self.model.setHeaderData(3, QtCore.Qt.Horizontal, "Street")
    self.model.setHeaderData(4, QtCore.Qt.Horizontal, "Street no.")
    self.model.setHeaderData(7, QtCore.Qt.Horizontal, "Reported by")

    # Hide the columns I don't want to see on the table view
    self.ui.tableIncidents.setModel(self.model)
    self.ui.tableIncidents.hideColumn(0)
    self.ui.tableIncidents.hideColumn(5)
    self.ui.tableIncidents.hideColumn(6)
    self.ui.tableIncidents.hideColumn(8)
```

When the user clicks the 'Clear Filter' button, the filter date range is reset and the 'Latest Incidents' QTableView is reloaded.

```
def clearIncidentFilter(self):
    # Disable the update tab because incident is not selected anymore
    self.ui.tabWidgetIncidents.setCurrentIndex(0)
    self.ui.tabWidgetIncidents.setTabEnabled(1, False)

    # Call function to reset the date range and reload the incidents table
    self.setIncidentFilterRangeDates()

def setIncidentFilterRangeDates(self):
    # Set 'to' value in range to current date
    self.ui.dateIncidentFilterTo.setDateTime(QtCore.QDateTime.currentDateTime())

    # Set 'from' value in range to one week ago
    one_week_ago = datetime.today() - timedelta(days=7)
    self.ui.dateIncidentFilterFrom.setDateTime(one_week_ago)

    # Reload the incidents table view
    self.populateIncidentTableView()
```

5. Database application – Current testing and use

Initially I just tested on my own while developing the application. A former colleague of mine is a member of the Table View Neighbourhood Watch, so it was useful to bounce ideas off him. After developing the user interface in PyQt, I first implemented the "perfect data" version of the code that would interact with the database. I then tried to cause errors by inputting incorrect data via the GUI. For each of those errors I implemented validation in the Python code and built up a validation message to display to the user.

I also tried to click on other buttons in different windows after performing certain actions to see if I could cause some errors. For example, after updating a member and filtering the member list, I found that clicking on the update button again would cause an error. This is because the button function relies on having a row of the table selected. I fixed this by disabling the 'Update' tab unless a row is selected.

I also check the database first on some of the insertion queries to see if the same thing already exists – for example for 'Available Patrol Slots' – to prevent duplicates in the database.

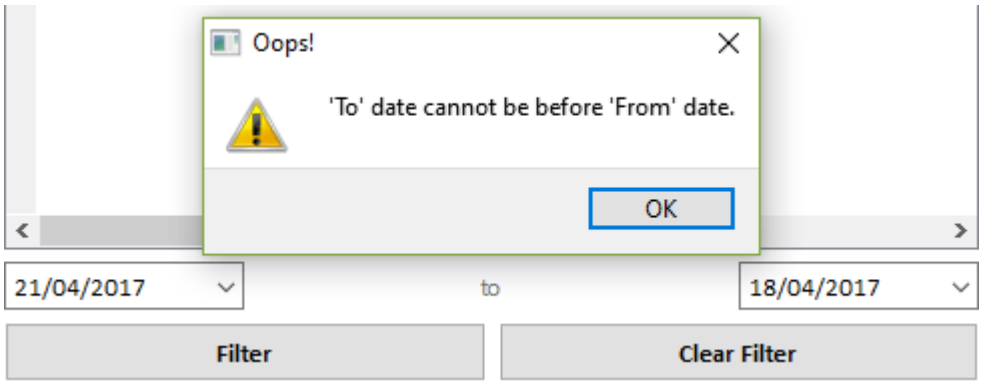
I then gave the application to the previously mentioned member of the TVNW and he found some further bugs (like being able to insert a really long phone number that the database field didn't cater for) which I fixed.

At the moment the application is not being used but they did express some interest in the idea.

Here are some examples of the validation I implemented after testing:

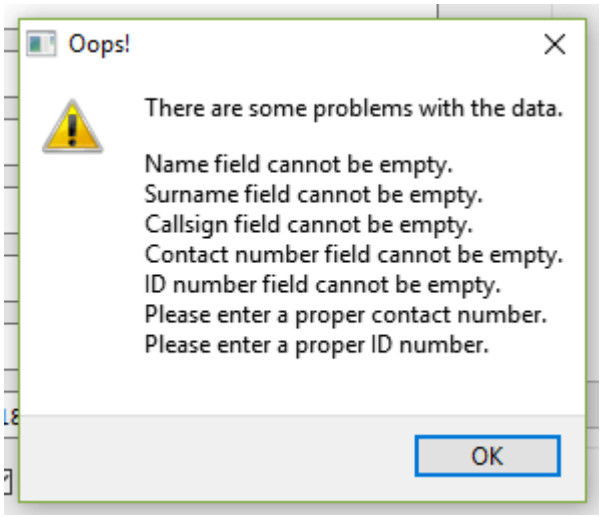
On the filter date range of the 'Latest Incidents' table:

```
    ...
    if to_date < from_date:
        showMessageBox(QMessageBox.Warning, "'To' date cannot be before 'From'
date.", "", "Oops!")
        return False
    ...
```



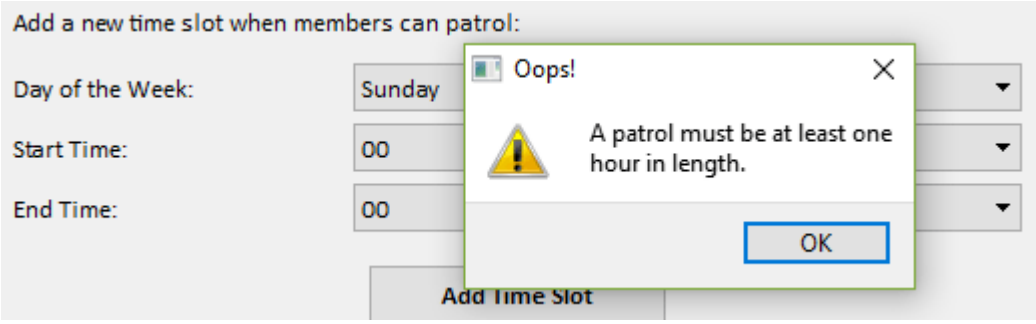
When adding a new member:

```
    ...
    validation_message = ""
    if name == "":
        validation_message += "Name field cannot be empty.\n"
    if surname == "":
        validation_message += "Surname field cannot be empty.\n"
    if callsign == "":
        validation_message += "Callsign field cannot be empty.\n"
    if contact_number == "":
        validation_message += "Contact number field cannot be empty.\n"
    if id_number == "":
        validation_message += "ID number field cannot be empty.\n"
    if name.isdigit():
        validation_message += "Please enter a proper name.\n"
    if surname.isdigit():
        validation_message += "Please enter a proper surname.\n"
    if not str(contact_number).isdigit():
        validation_message += "Please enter a proper contact number.\n"
    if (not str(id_number).isdigit()) or (not len(str(id_number)) == 13):
        validation_message += "Please enter a proper ID number.\n"
    if (not len(str(contact_number)) < 11):
        validation_message += "Contact number is too long.\n"
    if not (validation_message == ""):
        showMessageBox(QMessageBox.Warning, "There are some problems with the
data.", validation_message, "Oops!")
        return False
    ...
```



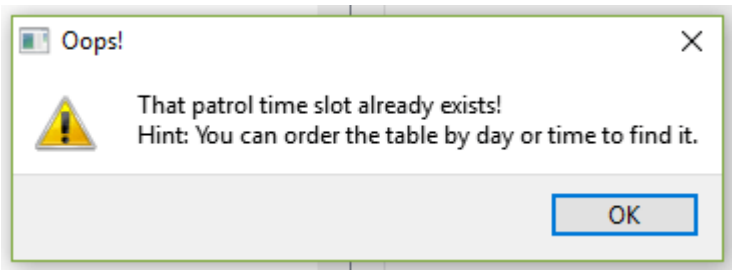
When adding a new patrol time slot:

```
    ...
    if start_hour == end_hour:
        validation_message = "A patrol must be at least one\nhour in length."
    if not (validation_message == ""):
        showMessageBox(QMessageBox.Warning, validation_message, "", "Oops!")
        return False
    ...
```



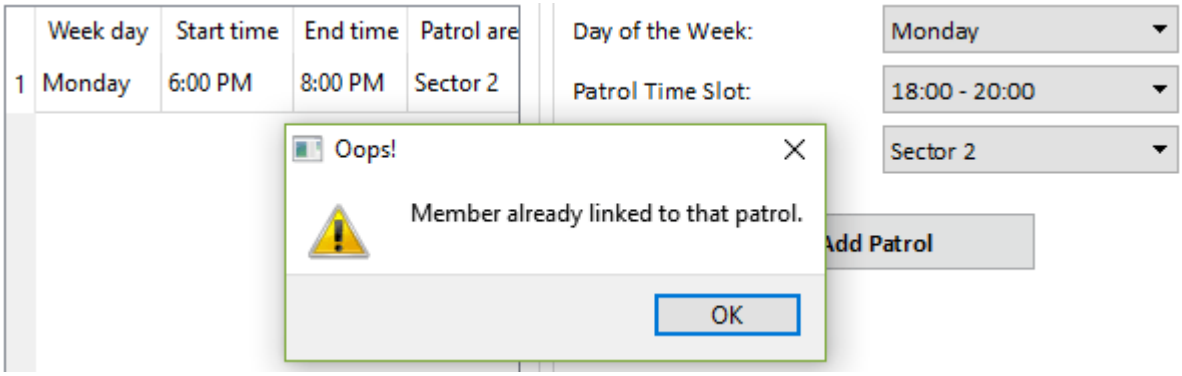
Checking if the new patrol time slot already exists:

```
    ...
    query = QtSql.QSqlQuery()
    query.prepare("SELECT (1) FROM patrol_time WHERE week_day = ? AND time_start
= ? AND time_end = ?")
    query.addBindValue(week_day)
    query.addBindValue(time_start)
    query.addBindValue(time_end)
    query.exec_()
    if query.size() > 0:
        showMessageBox(QMessageBox.Warning, "That patrol time slot already
exists!\nHint: You can order the table by day or time to find it.", "", "Oops!")
        return False
    ...
```



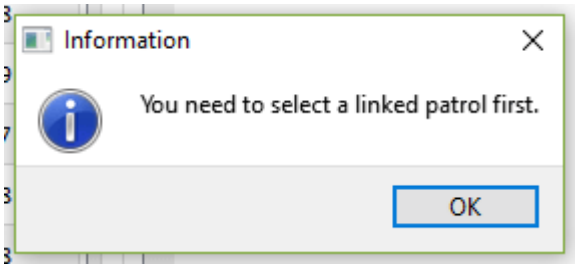
Checking to see if a member is already linked to a patrol time slot:

```
    ...
    query.prepare("SELECT * FROM patrol WHERE member_id = ? AND time_id = ? AND
area_id = ?")
    query.addBindValue(member_id)
    query.addBindValue(time_id)
    query.addBindValue(area_id)
    if not query.exec_():
        print(query.lastError().text())
    if query.size() > 0:
        # If a row exists, the member is already patrolling that time
        showMessageBox(QMessageBox.Warning, "Member already linked to that
patrol.", "", "Oops!")
    else:
    ...
```



When trying to remove a patrol from a member's schedule:

```
    ...
    if (self.ui.tableLinkedPatrols.currentIndex().row() == -1):
        msg = QMessageBox()
        msg.setIcon(QMessageBox.Warning)
        msg.setText("You need to select a linked patrol first.")
        msg.setWindowTitle("Information")
        msg.setStandardButtons(QMessageBox.Ok)
        msg.exec_()
        return False
    ...
```

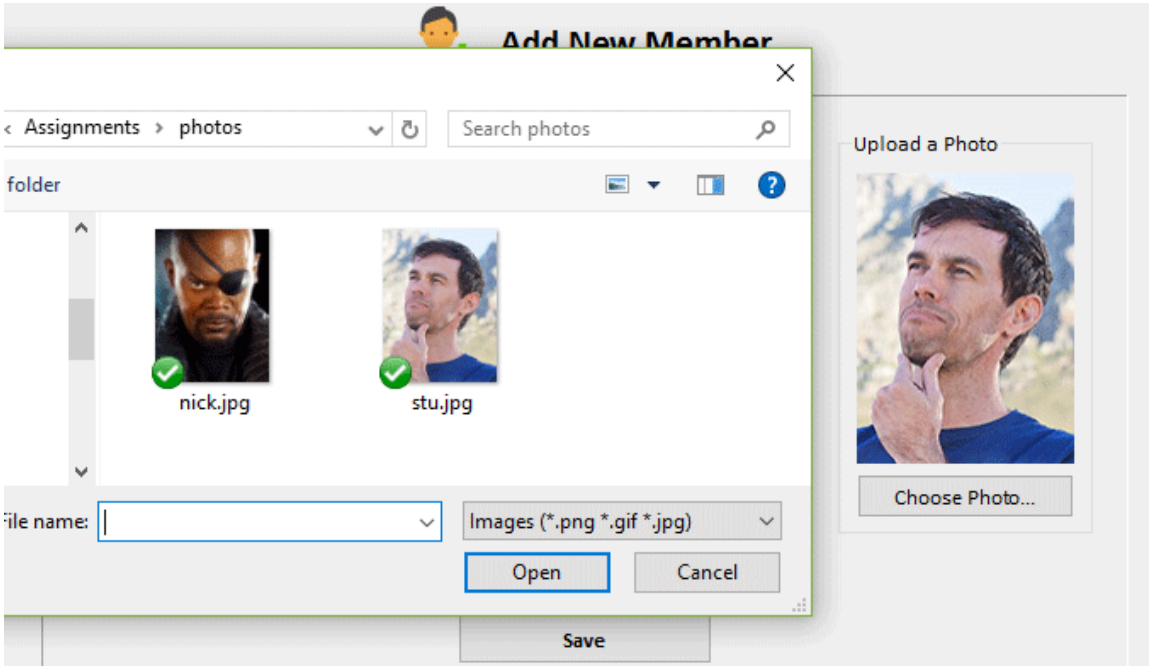


6. Enhancements

QPixmap and QFileDialog

I'm using QPixmap and QFileDialog widgets to load and display a member photo in a couple of places in the application (see the full code for functions `getPhotoPath()` and `addMemberPhoto()` on page 20).

```
...
self.ui.labelAddMemberPhoto.setPixmap(QtGui.QPixmap(filename))
self.ui.labelAddMemberPhoto.setScaledContents(True)
...
```



QWebView

I'm using QWebView widget to load https://www.gps-coordinates.net so that getting GPS coordinates for addresses are easy to obtain.

QSqlRelationalTableModel and QSqlRelation

I'm using these classes to be able to show the actual data behind the relations, for instance, to be able to show the incident description and member callsign in the 'Latest Incidents' table view.

QTime, QTimer and QLCDNumber

I'm using these widgets to show the current time and members patrolling at the current time.

```
    ...
    # Start timer for LCD time
    timer = QtCore.QTimer(self)
    timer.timeout.connect(self.showLcd)
    timer.start(1000)
    self.showLcd()

    # Start timer for list of patrolling members (updated every min)
    timerUpdateList = QtCore.QTimer(self)
    timerUpdateList.timeout.connect(self.populateCurrentlyPatrollingList)
    timerUpdateList.start(60000)
    self.populateCurrentlyPatrollingList()

def showLcd(self):
    time = QtCore.QTime.currentTime()
    self.ui.lcdNumber.display(time.toString('hh:mm'))
    ...
```

Thanks for reading...

The application code and other files can be downloaded here:
https://www.dropbox.com/s/uisq01aqcihol5c/Assignment2.zip?dl=0

It includes sql.txt which can be used to create the database.