Aperçu de quelques modules Qt.

François Delobel

Master TechMed et Setsys, Université Clermont-Auvergne

Plan

Solutions de stockage de données

JSON

- Syntaxe *human-friendly*.
- Fichiers légers
- Standard
- Accès direct pas possible

SQLite

- Base de données en un seul fichier
- Pas besoin de serveur, de gestions des utilisateurs
- Standard

XML

- Syntaxe plus lourde que JSON mais plus riche
- Mais possibilité de vérification de la syntaxe des fichiers
- Pas abordé ici (mais possible avec Qt)

Principes

Support de base pour **`bool`**, **`double`**, `string`, `array`, `object`, `null`.

`QJsonDocument`

- Gère l'accès (lecture, écriture) au document (fichier) JSON.

`QJsonObject`

- Un objet JSON: list de paires clef/valeur
- Valeurs: `QJsonValue`
- Surcharge `[]`

`QJsonArray`

- Construire des arrays pour stocker en des listes en JSON.
- Il faut construire chacun des éléments au préalable.

```
class Character
{
public:
    enum ClassType { Warrior, Mage, Archer };

    Character();
    Character(const QString &name, int level, ClassType classType);

    QString name() const;
    void setName(const QString &name);

    int level() const;
    void setLevel(int level);

    ClassType classType() const;
    void setClassType(ClassType classType);

    void read(const QJsonObject &json);          // Ici !!
    void write(QJsonObject &json) const;         // Et la!
private:
    QString mName;
    int mLevel;
    ClassType mClassType;
};
```

Affectation depuis un objet JSON

- On utilise l'accès en mode *map* (`[]`).
- On récupère un `QVariant`
- On le convertit dans le bon type.

```
void Character::read(const QJsonObject &json)
{
    mName = json["name"].toString();
    mLevel = json["level"].toDouble();
    mClassType = ClassType(qRound(json["classType"].toDouble()));
}
```

Écriture dans un objet JSON

```
void Character::write(QJsonObject &json) const
{
    json["name"] = mName;
    json["level"] = mLevel;
    json["classType"] = mClassType;
}
```

Une classe avec une liste

```
class Level
{
public:
    Level();

    // Un level contient une liste de
    // Non Players Characters (NPC)
    const QList<Character> &npcs() const;
    void setNpcs(const QList<Character> &npcs);

    void read(const QJsonObject &json);
    void write(QJsonObject &json) const;
private:
    QList<Character> mNpcs;
};
```

Extraction depuis une QJsonArray

```
void Level::read(const QJsonObject &json)
{
    mNpcs.clear();

    //On recupere l'Array
    QJsonArray npcArray = json["npcs"].toArray();

    // On la parcourt
    for (int npcIndex = 0; npcIndex < npcArray.size(); ++npcIndex) {

        // On recupere les QObjects en JSON un par un
        QJsonObject npcObject = npcArray[npcIndex].toObject();

        Character npc;
        // On extrait les donnees.
        npc.read(npcObject);
        mNpcs.append(npc);
    }
}
```

Construction d'une QJsonArray

```
void Level::write(QJsonObject &json) const
{
    // On construit une JArray vide
    QJsonArray npcArray;

    foreach (const Character npc, mNpcs) {

        // On cree un JObject vide
        QJsonObject npcObject;

        // On le remplit avec un perso
        npc.write(npcObject);
        // On ajoute le JObject a la JArray
        npcArray.append(npcObject);
    }
    json["npcs"] = npcArray;
}
```

Ouvrir un QJsonDocument

```
bool Game::loadGame(Game::SaveFormat saveFormat) {
    // Choix entre fichier texte et binaire
    QFile loadFile(saveFormat == Json
        ? QStringLiteral("save.json")
        : QStringLiteral("save.dat"));

    if (!loadFile.open(QIODevice::ReadOnly)) {
        qWarning("Couldn't open save file.");
        return false;
    }
    // On recupere tout le contenu du fichier
    QByteArray saveData = loadFile.readAll();
    // On charge le QJsonDocument
    QJsonDocument loadDoc(saveFormat == Json
        ? QJsonDocument::fromJson(saveData)
        : QJsonDocument::fromBinaryData(saveData));

    read(loadDoc.object());
    return true;
}
```

Écrire un fichier QJsonDocument

```
bool Game::saveGame(Game::SaveFormat saveFormat) const {
    QFile saveFile(saveFormat == Json
        ? QStringLiteral("save.json")
        : QStringLiteral("save.dat"));

    if (!saveFile.open(QIODevice::WriteOnly)) {
        qWarning("Couldn't open save file.");
        return false;
    }
    // On cree l'objet JSON
    QJsonObject gameObject;
    write(gameObject);
    QJsonDocument saveDoc(gameObject); // On cree un JDocument
    saveFile.write(saveFormat == Json
        ? saveDoc.toJson()              // On ecrit le flux
        : saveDoc.toBinaryData());

    return true;
}
```

Principes

- Gestion unifiée de plusieurs bases de données:
  - SQLite,
  - PostgreSQL,
  - MariaDB/MySQL,
  - IBM DB2,
  - Borland,
  - Oracle,
  - OBDC (Microsoft).
- Dans le cas de Sailfish, souvent usage de *SQLite* (BD dans un seul fichier) pour stockage local.

Trois niveaux d'intégration

Drivers

- Gèrent la base de donnée proprement dite et en présentent un aspect unifié aux autres couches.
- On choisit le bon driver et OSEF!

API SQL

- Les classes et méthodes pour envoyer des requêtes, lire des résultats, ouvrir une connexion...

Modèle SQL

- Pour les afficher directement dans les vues :)

J'ouvre ma base données

- J'ai besoin d'une seule connexion.
  - Je l'ouvre sans lui donner de nom, c'est la BD par défaut.
- J'ai besoin de plusieurs connexions
  - Je passe un nom en deuxième argument de addDataBase.

```
QSqlDatabase db = QSqlDatabase::addDatabase("QPSQL"); // PostgreSQL
db.setHostName("acidalia");          // Machine cible
db.setDatabaseName("customdb");    // Nom de la BD
db.setUserName("mojito");          // User
db.setPassword("J0a1m8");         // Pass
bool ok = db.open();              // Test
```

Je la requête et je parcours

- Ici, sur la base de données par défaut.
- La requête contient son résultat après exécution.
- Changement d'enregistrement avec `next()`.
- Accès aux champs via un index qui commence à 0.

```
QSqlQuery query("SELECT country FROM artist");
while (query.next()) {
    QString country = query.value(0).toString();
    doSomething(country);
}
```

Query avec *placeholders*

- On prépare la requête avec des *placeholders*.
- On donne des valeurs aux placeholders.
- On exécute la requête.

```
QSqlQuery query;
query.prepare("INSERT INTO person (id, forename, surname) "
              "VALUES (:id, :forename, :surname)");
query.bindValue(":id", 1001);
query.bindValue(":forename", "Bart");
query.bindValue(":surname", "Simpson");
query.exec();
```

Les BDs comme modèle

- Requêtes comme modèle.
- Table comme modèle.

```
QSqlQueryModel *model = new QSqlQueryModel;
model->setQuery("SELECT name, salary FROM employee");
model->setHeaderData(0, Qt::Horizontal, tr("Name"));
model->setHeaderData(1, Qt::Horizontal, tr("Salary"));

QTableView *view = new QTableView;
view->setModel(model);
view->show();
```

Sauver un fichier de configuration avec QSettings

- Outil QML de stockage standard, indépendant de la plateforme.
  - Fichiers sous unix ou mac
  - Base de registre sous Windows
  - Peut même générer des fichiers `.ini`
- Peut stocker tout `QVariant`.
- Supporte des *sections* (`section/element`).
- Par défaut bien initialisé par `libSailfish` sauf si usage des jails.

```
QSettings settings;
settings.setValue("editor/wrapMargin", 42);
settings.sync(); // Force la sauvegarde

int margin = settings.value("editor/wrapMargin").toInt();

// Retourne 80 si pas de valeur par defaut
int margin = settings.value("editor/wrapMargin", 80).toInt();
```

Où stocker mes informations?

Technique: demander à `QStandardPaths`!

- Pour les données persistantes de l'application
  - `QStandardPaths::writableLocation(QStandardPaths::DataLocation)`
- Pour les données *en cache* (qui peuvent être supprimées):
  - `QStandardPaths::writableLocation(QStandardPaths::CacheLocation)`

Retournent un chemin sous forme de `QString`. Si il n'existe pas, il faut le créer.

Principes du Drag & Drop: décollage

`MouseArea`: la zone qui gère la souris

- Possédée par un composant graphique (qui ne doit pas être ancré sus ses axes de *drag*
- `drag.active: true` par défaut.

Les propriétés de `drag` (dans la `MouseArea`)

- `drag.target`: l'objet qui doit bouger (un parent)
- `drag.axis`: l'axe de déplacement autorisé (e.g., `Drag.YAxis`)
- Les amplitudes de déplacement (e.g., `drag.maximumY`)
- Un appel à la fonction `Drag.drop()` de la cible au `onReleased` de la souris
- Une redéfinition du `hotSpot` (souvent à la position de la souris)
- `drag.active` est vrai si un *drag* est en cours.

Principes du Drag & Drop: largage

La zone de largage `DropArea`

- Reçoit les éléments largués.
- Propriété `drag` avec `drag.source`
- `containsDrag`: vrai si des éléments dragués dans la zone.
- Signaux:
  - `dropped(DragEvent)` quand un élément est posé
  - `entered(DragEvent)` quand un élément survole.
- `key`: liste pour filtrer les drops.

```
Rectangle {   width: 300; height: 300

    Rectangle {
        x: 10; y: 210;          width: 50; height: 50
        color: "blue"
        Drag.active: mouseArea.drag.active
        Drag.keys: "blue"
        MouseArea {
            id: mouseArea
            anchors.fill: parent
            drag.target: parent
            onReleased: parent.Drag.drop()
            onPressed: parent.Drag.hotSpot = Qt.point(mouse.x, mouse.y)
        }
    }
    DropArea {
        x: 210; y: 10;          width: 50; height: 50
        onDropped: console.debug("onDropped")
        keys: "blue"
        Rectangle {
            anchors.fill: parent
            border.color: parent.containsDrag ? "darkgrey" : "lightgrey"
            color: parent.containsDrag ? "lightgrey" : "white"
        }
    }
}
```

Lire n'importe quelle touche

```
Rectangle {
    focus: true // Si pas focus, pas touche!
    Keys.onPressed: {
        if ( event.key == Qt.Key_Left ) {
            console.log("À gauche toute!");
            event.accepted = true; // Pour eviter de remonter aux parents
        }
    }
}
```

Gestion de la validation par clavier (virtuel)

`EnterKey` accessible directement

```
TextField {
    // ...
    EnterKey.onClicked: number=text
}
```

Choix du clavier virtuel et Contrôle de saisie

```
TextField {
        x: Theme.horizontalPageMargin
        inputMethodHints: Qt.ImhDigitsOnly // Choix pavé numérique
        validator: IntValidator { bottom: 0; top: 140;}            // Contrô
        label: "Age"
        placeholderText: "Between 0 -> 140"
        font.pixelSize: Theme.fontSizeExtraLarge
        width: parent.width
}
```

Propriétés des claviers `Qt::InputMethodHint`

Comportement:

- `Qt::ImhHiddenText` (voir aussi `QLineEdit::echoMode=Password`)
- `Qt::ImhNoAutoUppercase`
- `Qt::ImhDate`
- `Qt::ImhTime`
- ...

Type de contenu:

- `Qt::ImhDigitsOnly` Entiers
- `Qt::ImhFormattedNumbersOnly` Flottants
- `Qt::ImhUppercaseOnly` Majuscules
- `Qt::ImhLowercaseOnly` Minuscules
- `Qt::ImhDialableCharactersOnly` Numéro Téléphonique
- `Qt::ImhEmailCharactersOnly` email
- `Qt::ImhUrlCharactersOnly` URL
- `Qt::ImhLatinOnly` À éviter

Validateurs

Validateurs QML (qui s'appuient sur les validateur Qt/C++)

- `IntValidator` (bottom, top)
- `DoubleValidator` (bottom, top, decimals, notation)
- `RegExpValidator` (regExp)

Faire un *timer* en QML (ou QTimer en C++)

```
Rectangle {
    width: 350; height: 50
    Component.onCompleted: timer.start()
    property int count: 0
    Timer {
        id: timer
        interval: 1000
        repeat: true
        onTriggered: count = count + 1
    }
    Text {
        anchors.centerIn: parent
        font.pointSize: 16
        text: "L'application a demarre depuis " + count + " s"
    }
}
```

Fin d'une application

- À la mort de l'application, l'`ApplicationWindow` est détruite.
- Si besoin d'exécuter du code à la fermeture:
  - `Component.onDestruction`!

Afficher une page web

- Permet d'afficher tout ou partie.
- QML: `WebView`
- Importer `QtWebKit 3.0`
  - Comportement différent Sailfish et dernières version de Qt

```
Page {
    Rectangle  {
        anchors.fill: parent

        WebView {
            id: webview

            url: "http://qt-project.org"
            anchors.fill: parent
        }
    }
}
```

États d'un élément

- Tout élément (`Item`) est exactement dans un état (`""` par défaut).
  -
  - Pour les objets qui ne sont pas des éléments, voir `StateGroup`
- `state` contient l'état courant.
- Il peut en définir plusieurs.
  - `states` contient les états possibles.
- Des actions peuvent être conditionnées à cet état

State: un état

- `name` le nom de l'état
- `PropertyChanges`: des *map* de changement de propriété quand on entre dans cet état.
  - `target`: *id* de l'objet dont on veut changer les propriétés.
  - Lites des couples `propriete: valeur`
- `StateChangeScript`: exécute un script au changement d'état
- `ParentChange` pour faire changer le père!
- `AnchorChanges` pour modifier les ancres.

Exemple de définition d'état et de changement de propriété

```
Rectangle {
    id: signal
    width: 200; height: 200
    state: "NORMAL"

    states: [
        State {
            name: "NORMAL"
            PropertyChanges { target: signal; color: "green"}
            PropertyChanges { target: flag; state: "FLAG_DOWN"}
        },
        State {
            name: "CRITICAL"
            PropertyChanges { target: signal; color: "red"}
            PropertyChanges { target: flag; state: "FLAG_UP"}
        }
    ]
}
```

Exemple de changement d'état sur signal

```
Rectangle {
    id: signalswitch
    width: 75; height: 75
    color: "blue"

    MouseArea {
        anchors.fill: parent
        onClicked: {
            if (signal.state == "NORMAL")
                signal.state = "CRITICAL"
            else
                signal.state = "NORMAL"
        }
    }
}
```

Changer d'état sur changement de condition

```
Rectangle {
    id: bell
    width: 75; height: 75
    color: "yellow"

    states: State {
                name: "RINGING"

                // Quand la condition devient vraie...
                when: (signal.state == "CRITICAL")
                PropertyChanges {target: speaker; play: "RING!"}
            }
}
```

```
Rectangle {
    width: 75; height: 75
    id: button
    state: "RELEASED"
    MouseArea {
        anchors.fill: parent
        onPressed: button.state = "PRESSED"
        onReleased: button.state = "RELEASED"
    }

    states: [
        State {
            name: "PRESSED"
            PropertyChanges { target: button; color: "lightblue"}
        },
        State {
            name: "RELEASED"
            PropertyChanges { target: button; color: "lightsteelblue"}
        }
    ]

    transitions: [
        Transition {
            from: "PRESSED"
            to: "RELEASED"
            ColorAnimation { target: button; duration: 100}
        },
        Transition {
            from: "RELEASED"
            to: "PRESSED"
            ColorAnimation { target: button; duration: 100}
        }
    ]
}
```