Using the Open-Transactions API and UI models

Version 1.0
5/29/2022

UI flow for a generic Open-Transactions wallet

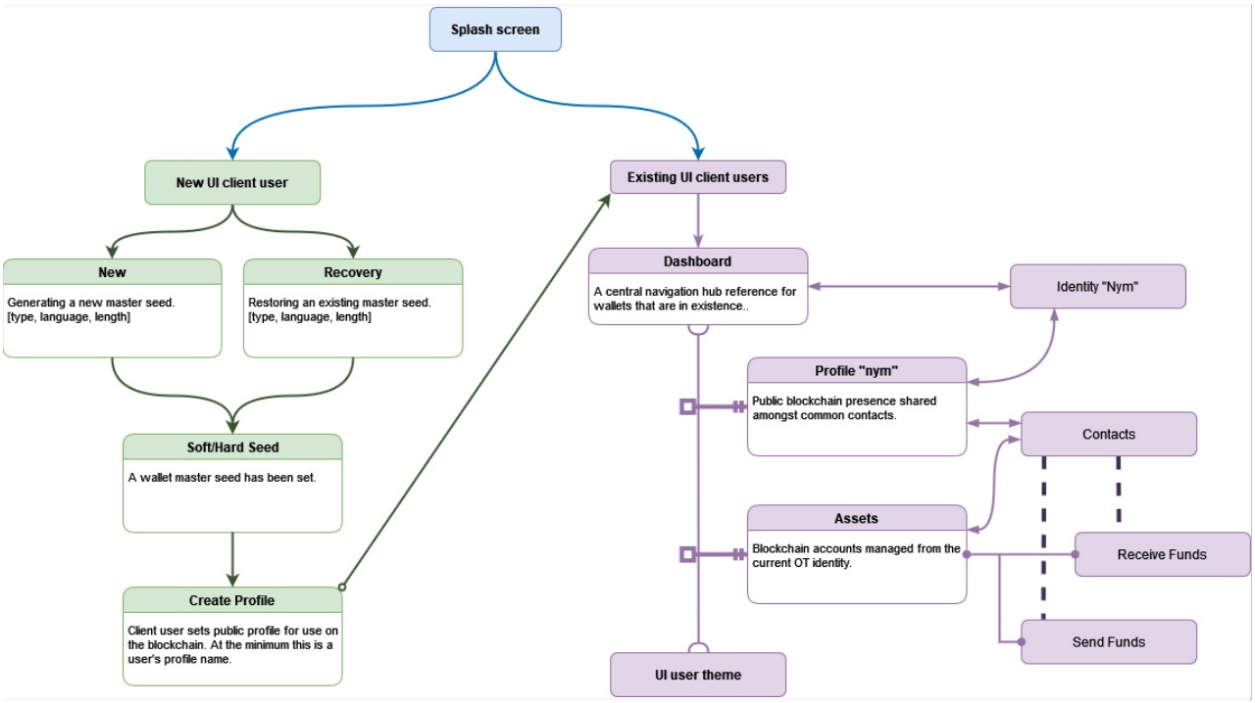


Figure 1 - Generic wallet UI user flow

In figure 1, the green diagram on the left shows the UI startup sequence for the wallet.
The purple diagram on the right represents the UI flow during normal operation of the wallet.

Reference Implementation

Open-Transactions (*OT* or *opentxs*) provides a reference implementation for a Qt-based wallet UI written in C++. This wallet project is called “Metier” and the source code is located on GitHub at this URL:
https://github.com/Open-Transactions/metier

For those who have permission to see it, MatterFi’s fork of this project, including multiple QML interfaces, is located here:
https://github.com/matterfi/wallet

Code samples in this document are all excerpted from the above C++ reference implementation.

Wallet UI models

Open-Transactions provides UI models and QModels for use with Qt.

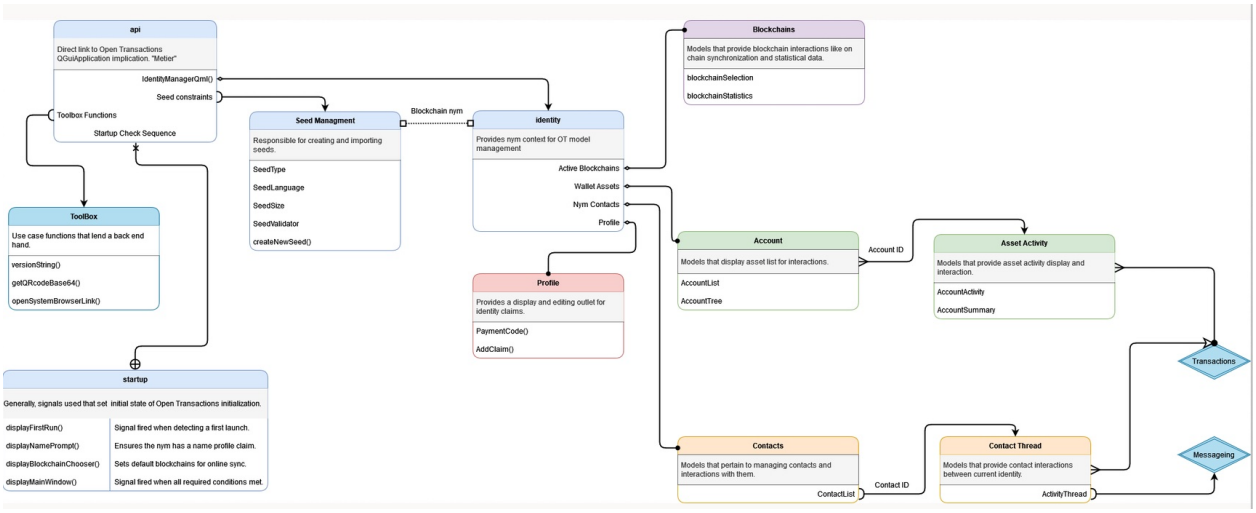


Figure 2 – On the left side is a diagram of the Metier wallet's high-level OT wrapper, based on Qt Signals and Slots. On the right side are QModels from OT that are relevant to wallet developers and are described in further detail in the below documentation.

When running metier using the --advanced command line option, it uses **QWidgets** in the UI. But in QML, the UI objects are **QObjects**, as seen in figure 3:

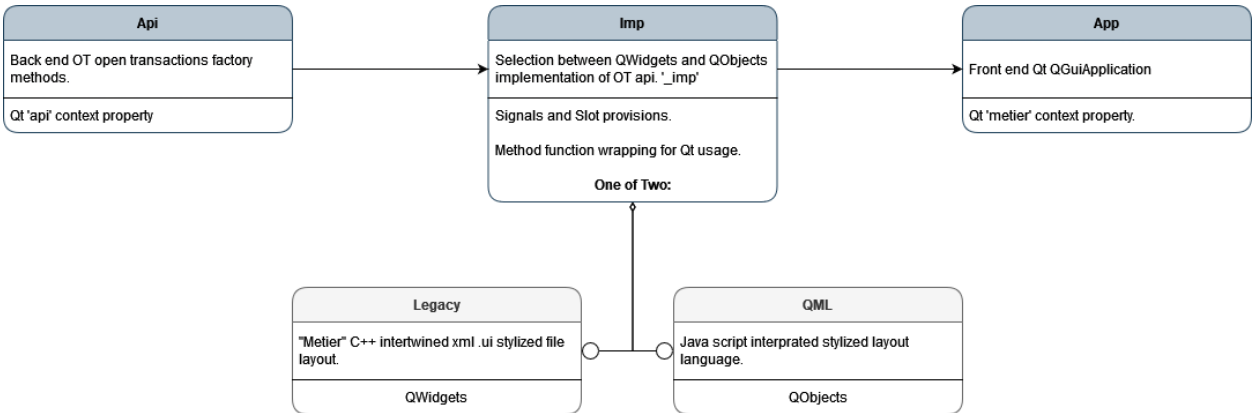


Figure 3 – QWidgets for the C++ UI, QObjects for the QML UI

- Most of the Qt model class headers are located here:
  "metier/deps/opentxs/include/opentxs/interface/qt/"
- All QML design work is done within this directory only:
  "metier/src/skin/qml/"
- The QWidgets used in the advanced Qt C++ UI "Metier" are here:
  "metier/src/widgets/"

Error codes

Most of the OT API is *noexcept* and so very few exceptions can possibly be thrown. OT API functions often return bool, allowing the developer to check for success or failure. If a function returns a pointer, check for null. If it instead returns an OT object, the object may be a “blank object” signifying failure. In that case, check for version >0.

Startup process

- **"metier/src/app/*.[cpp|hpp]",**
- **“metier/src/api/*.[cpp|hpp]”**

When starting up an OT wallet application, the developer must first instantiate an OT context object, followed by a client (wallet) session object, as shown below.

Our wallet reference implementation in C++ has a main singleton with a few member variables including the OT context object *ot_* and the client (wallet) session object *api_*. As seen here:

```
PasswordCallback callback_;
opentxs::PasswordCaller caller_;
const opentxs::api::Context& ot_;
const ot::OTZMQListenCallback rpc_cb_;
ot::OTZMQRouterSocket rpc_socket_;
const opentxs::api::session::Client& api_;
```

The **opentxs context** member variable in the above code is named *ot_*. This is the only OT context that our reference implementation instantiates, but it is possible when using the OT library to instantiate multiple contexts and to have them all initialized and running simultaneously. It’s also possible to have multiple **client sessions** and **server sessions** running simultaneously. The OT unit tests make use of this capability.

Here is our reference implementation’s C++ code for instantiating the *ot_* context object:

```
, callback_(app)
, caller_()
, ot_(ot::InitContext(
      make_args(parent, argc, argv),
      [this]() -> auto{
          caller_.SetCallback(&callback_);
          return &caller_;
      }()))
```

The *ot_* object is used for various Open-Transactions function calls that exist independent of, and prior to the existence of, a client or server session. The password callback used above is optional and allows the wallet developer to request a password from the user when creating or using a private key.

In our example, the **client session** is an object named *api_*. Below is C++ code showing how that *api_* object is initialized:

```
, api_(ot_.StartClientSession(ot_args_, 0))
```

The parameter *ot_args_* passed to the above *StartClientSession()* call is of type *ot::Options* and is initialized from the command line like this:

```
static const auto ot_args_ = ot::Options{};

static auto make_args(QGuiApplication& parent, int& argc, char** argv)
    noexcept
    -> const ot::Options&
{
    parent.setOrganizationDomain(Api::Domain());
    parent.setApplicationName(Api::Name());
    auto& args = const_cast<ot::Options&>(ot_args_);
    args.ParseCommandLine(argc, argv);
```

When starting up for the first time, the wallet must be initialized. The first thing our reference implementation does is check to see if a seed already exists, which it does using its *doNeedSeed()* function:

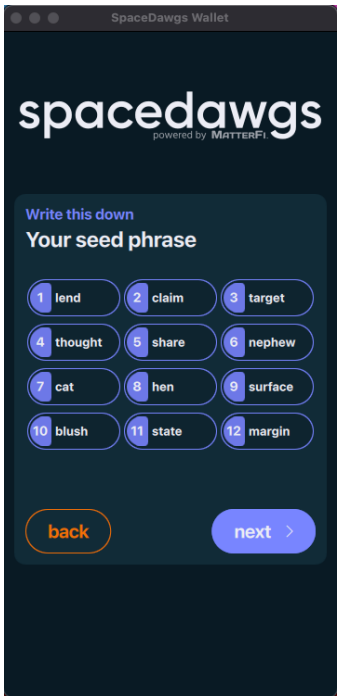
```
auto Api::doNeedSeed() -> void
{
    switch (imp_.state_.load()) {
        case Imp::State::init: {
            const auto [seed, count] = imp_.api_.Crypto().Seed().DefaultSeed();

            if (seed.empty() && (0u == count)) {
                emit privateNeedSeed({});
            } else {
                if (false == imp_.validateSeed()) {
                    qFatal("Unable to initialize wallet seed");
                }

                imp_.state_.store(Imp::State::have_seed);
                emit checkStartupConditions();
            }
        } break;
        case Imp::State::have_seed:
        case Imp::State::have_nym:
        case Imp::State::run:
        default: {
        }
    }
}
```

The above OT API call *api_.Crypto().Seed().DefaultSeed()* grabs the default seed, to see if one even exists in the wallet.

The return value is verified using the above code: *if(seed.empty() && (0u == count)).* This is how the wallet developer determines whether or not the wallet still needs to generate (or import) a seed before continuing to the main wallet UI.

The function *createNewSeed()* is used in our reference implentation for actually creating the new wallet seed and demonstrates use of the OT API call *api_.Crypto().Seed().NewSeed()* seen below:

```
auto createNewSeed(
    const int type,
    const int lang,
    const int strength) noexcept -> QStringList
{
    ready().get();
    wait_for_seed_backup_ = true;
    auto lock = Lock{lock_};
    auto success{false};
    auto& id = const_cast<std::string&>(seed_id_);
    auto postcondition = ScopeGuard{[&]() {
        if (false == success) { id = {}; }
    }};
    const auto& seeds = api_.Crypto().Seed();

    assert(id.empty());
    assert(0 == seeds.DefaultSeed().second);

    const auto invalid = [](const int in) -> auto
    {
        return (0 > in) || (std::numeric_limits<std::uint8_t>::max() < in);
    };

    if (invalid(type) || invalid(lang) || invalid(strength)) { return {}; }

    auto reason =
        api_.Factory().PasswordPrompt("Generate a new Metier wallet seed");
    id = seeds.NewSeed(
        static_cast<ot::crypto::SeedStyle>(static_cast<std::uint8_t>(type)),
        static_cast<ot::crypto::Language>(static_cast<std::uint8_t>(lang)),
        static_cast<ot::crypto::SeedStrength>(
            static_cast<std::size_t>(strength)),
        reason);

    if (id.empty()) { return {}; }

    auto notUsed{false};
    const auto config = api_.Config().Set_str(
        ot::String::Factory(
            QGuiApplication::applicationName().toStdString()),
        ot::String::Factory(seed_id_key),
        ot::String::Factory(id),
        notUsed);

    if (false == config) { return {}; }
    if (false == api_.Config().Save()) { return {}; }

    success = true;
    const auto words = QString{seeds.Words(id, reason).c_str()};

    return words.split(' ', Qt::SkipEmptyParts);
}
```

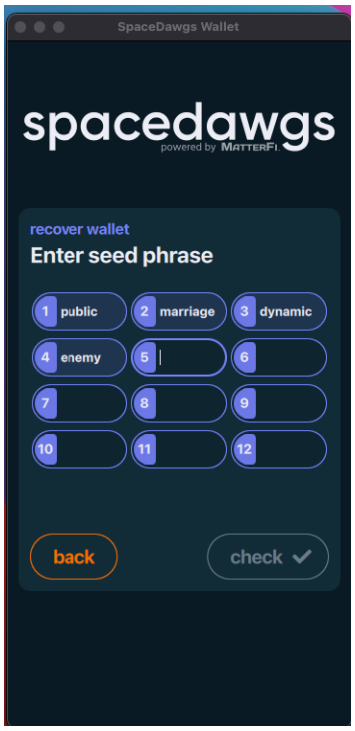


**This screenshot is only an example.*

The above function returns a QStringList containing the actual seed words for display in the UI as seen in the above example screenshot.

Our reference implementation calls *createNewSeed()* (the function above) whenever a new seed is needed. Otherwise it uses *importSeed()* (seen below) to restore the seed from a backup of the seed words:

```
auto importSeed(
    int type,
    int lang,
    const QString& input,
    const QString& password) -> void
{
    ready().get();
    auto lock = Lock{lock_};
    auto success{false};
    auto& id = const_cast<std::string&>(seed_id_);
    auto postcondition = ScopeGuard{[&]() {
        if (false == success) { id = {}; }
    }};
    const auto& seeds = api_.Crypto().Seed();

    assert(id.empty());
    assert(0u == seeds.DefaultSeed().second);

    auto reason =
        api_.Factory().PasswordPrompt("Import a Metier wallet seed");
    const auto words = api_.Factory().SecretFromText(input.toStdString());
    const auto passphrase =
        api_.Factory().SecretFromText(password.toStdString());
    id = seeds.ImportSeed(
        words,
        passphrase,
        static_cast<ot::crypto::SeedStyle>(static_cast<std::uint8_t>(type)),
        static_cast<ot::crypto::Language>(static_cast<std::uint8_t>(lang)),
        reason);

    if (id.empty()) { return; }

    auto notUsed{false};
    const auto config = api_.Config().Set_str(
        ot::String::Factory(
            QGuiApplication::applicationName().toStdString()),
        ot::String::Factory(seed_id_key),
        ot::String::Factory(id),
        notUsed);

    if (false == config) { return; }
    if (false == api_.Config().Save()) { return; }

    success = true;
}
```

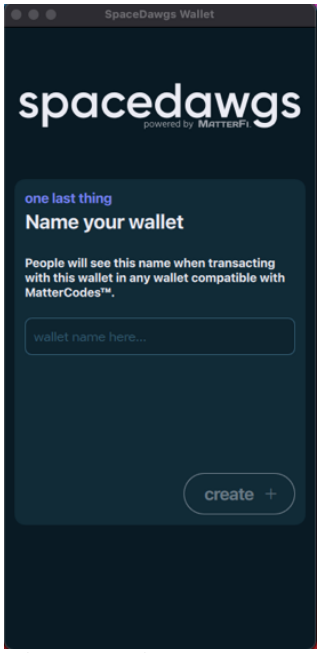


*This screenshot is only an example.

Once a seed is available, our reference implementation ensures that there is at least one Nym (identity) in the wallet before proceeding to the main window. Our reference implementation calls *api_.wallet().DefaultNym()* to determine whether or not a Nym exists, as seen in the example function *doNeedNym()* below.

If none exists and a Nym must therefore be created, the wallet application should ask the user to input an *alias*, or *display name* for that Nym. The developer should pass that name as the *alias* parameter to the *createNym()* call seen in the top line of code below:

```
auto Api::createNym(QString alias) -> void { imp_.createNym(alias); }

auto Api::doNeedNym() -> void
{
    switch (imp_.state_.load()) {
        case Imp::State::init: {
            doNeedSeed();
        } break;
        case Imp::State::have_seed: {
            const auto [nym, count] = imp_.api_.Wallet().DefaultNym();

            if (nym->empty()) {
                if (false == imp_.wait_for_seed_backup_) {
                    emit privateNeedProfileName({});
                }
            } else {
                if (false == imp_.validateNym()) {
                    qFatal("Unable to initialize identity");
                }

                imp_.state_.store(Imp::State::have_nym);
                emit checkStartupConditions();
            }
        } break;
        case Imp::State::have_nym:
        case Imp::State::run:
        default: {
        }
    }
}
```



*This screenshot is only an example.

Below is the actual code inside *createNym()* that our reference implementation uses to instruct OT to create a new Nym. The relevant line is code is:

const auto pNym = api_.Wallet().Nym({seed_id_, 0}, reason, alias.toStdString());

Check the return value for null, as seen in the below example:

```
auto createNym(QString alias) noexcept -> void
{
    ready().get();
    auto lock = Lock{lock_};
    const auto reason =
        api_.Factory().PasswordPrompt("Generate a new Metier identity");

    assert(false == seed_id_.empty());

    const auto pNym =
        api_.Wallet().Nym({seed_id_, 0}, reason, alias.toStdString());

    if (!pNym) { qFatal("Failed to create nym"); }

    const auto& nym = *pNym;
    bool notUsed{false};
    const auto config = api_.Config().Set_str(
        ot::String::Factory(
            QGuiApplication::applicationName().toStdString()),
        ot::String::Factory(nym_id_key),
        ot::String::Factory(nym.ID().str()),
        notUsed);

    if (false == config) { qFatal("Failed to update configuration"); }
    if (false == api_.Config().Save()) {
        qFatal("Failed to save config file");
    }

    identityManager()->setActiveNym(QString::fromStdString(nym.ID().str()));
    have_nym_ = true;
}
```

BlockchainSelection: Managing blockchains in the local wallet

"metier/deps/opentxs/include/opentxs/interface/qt/BlockchainSelection.hpp"

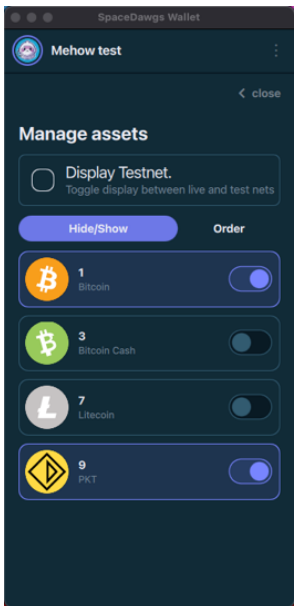
See model: opentxs::ui::BlockchainSelectionQt

To access the blockchain selection model, call:

```
opentxs::ui:: BlockchainSelectionQt * blockchains = api_.UI().BlockchainSelectionQt();
```

Below are the relevant functions in the BlockchainSelectionQt model. This model is a standard Qt::Model and works natively with Qt UI controls. Each row in this model represents a different blockchain. There is a Name column for UI display of the blockchain name for each row, and there are also informational roles available for each row as specified in *enum Roles* below:

```
class OPENTXS_EXPORT opentxs::ui::BlockchainSelectionQt final : public qt::Model
{
    Q_OBJECT
    Q_PROPERTY(int enabledCount READ enabledCount NOTIFY enabledChanged)

signals:
    void chainEnabled(int chain);
    void chainDisabled(int chain);
    void enabledChanged(int enabledCount);

public:
    /// chain is an opentxs::blockchain::Type, retrievable as TypeRole
    // NOLINTNEXTLINE(modernize-use-trailing-return-type)
    Q_INVOKABLE bool disableChain(const int chain) noexcept;
    /// chain is an opentxs::blockchain::Type, retrievable as TypeRole
    // NOLINTNEXTLINE(modernize-use-trailing-return-type)
    Q_INVOKABLE bool enableChain(const int chain) noexcept;

public:
    enum Roles {
        NameRole = Qt::UserRole + 0,   // QString
        TypeRole = Qt::UserRole + 1,   // int, opentxs::blockchain::Type
        IsEnabled = Qt::UserRole + 2,  // bool
        IsTestnet = Qt::UserRole + 3,  // bool
    };
    enum Columns {
        NameColumn = 0,
    };

    auto enabledCount() const noexcept -> int;
```



*This screenshot is only an example.

Bitcoin vs Ethereum Accounts

If the library is compiled to support Bitcoin, then BTC will be available as one of the blockchains in the Blockchain Selection model. If enabled, then a BTC account will appear as one of the wallet accounts in the AccountList model.

Similarly, if the library is compiled to support Ethereum, then ETH will be available as one of the blockchains in the Blockchain Selection model. If enabled, then an ETH account will appear as one of the wallet accounts in the AccountList model.

BTC and ETH accounts are otherwise identical from the perspective of the UI, and they appear side-by-side in the UI along with all of the other accounts in the wallet.

IdentityManager: Managing nyms in the local wallet

"metier/deps/opentxs/include/opentxs/interface/qt/IdentityManager.hpp"

See: opentxs::ui::IdentityManagerQt

To access the identity manager, call:

```
opentxs::ui::IdentityManagerQt* identity_manager  = api_.UI().IdentityManagerQt();
```

Here are the relevant functions in IdentityManagerQt:

```
auto getAccountActivity(const QString& accountID) const noexcept
    -> AccountActivityQt*;
auto getAccountList() const noexcept -> AccountListQt*;
auto getAccountStatus(const QString& accountID) const noexcept
    -> BlockchainAccountStatusQt*;
auto getAccountTree() const noexcept -> AccountTreeQt*;
auto getActiveNym() const noexcept -> QString;
auto getActivityThread(const QString& contactID) const noexcept
    -> ActivityThreadQt*;
auto getContactList() const noexcept -> ContactListQt*;
auto getNymList() const noexcept -> NymListQt*;
auto getNymType() const noexcept -> QAbstractListModel*;
auto getProfile() const noexcept -> ProfileQt*;

auto setActiveNym(QString) noexcept -> void;
```

Since the models returned from the above calls are QModels, they can be plugged directly into QWidgets and other Qt UI controls, and should work automatically.

The above functions relevant to your own user identity are *getNymList(), getActiveNym(), setActiveNym(),* and *getProfile()*. To retrieve the list of accounts for the active Nym, call *getAccountList().* To retrieve a history of activity for a specific account, call *getAccountActivity(accountId)*. Details on these calls are provided below. The models returned are all standard Qt models and can be plugged directly into Qt List Views and other UI controls.

User Profile

"metier/deps/opentxs/include/opentxs/interface/qt/Profile.hpp"

- See: opentxs::ui::ProfileQt

The profile contains the local wallet user's data. This model allows access to the local wallet user's display name, as well as his NymId and his Payment Code.

To get the profile, call:

```
opentxs::ui::ProfileQt* profile  = identity_manager->getProfile();
```

Since Qt uses pointers, be sure to check the return value of the above function for null:

```
if (null != profile) { … }
```

```
class OPENTXS_EXPORT opentxs::ui::ProfileQt final :
    public qt::Model
{
    Q_OBJECT
    Q_PROPERTY(QString displayName READ displayName
        NOTIFY displayNameChanged)
    Q_PROPERTY(QString nymID READ nymID CONSTANT)
    Q_PROPERTY(QString paymentCode READ paymentCode
        NOTIFY paymentCodeChanged)

signals:
    void displayNameChanged(QString) const;
    void paymentCodeChanged(QString) const;

public:
    // Tree layout
    QString displayName() const noexcept;
    QString nymID() const noexcept;
    QString paymentCode() const noexcept;
```

*This screenshot is only an example.

AccountList

"metier/deps/opentxs/include/opentxs/interface/qt/AccountList.hpp"

- See: opentxs::ui::AccountListQt

Each row in the AccountList represents a different account. The columns and Qt roles are provided below, showing the data available for each row.

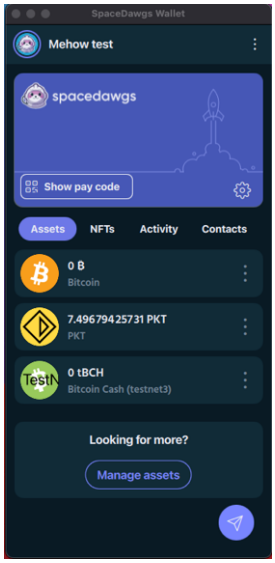
To access the account list, call:

```
opentxs::ui::AccountListQt* accounts  = identity_manager->getAccountList();
```

Since Qt uses pointers, be sure to check the return value of the above function for null:

```
if (null != accounts) { … }
```

```
class OPENTXS_EXPORT opentxs::ui::AccountListQt final : public qt::Model
{
    Q_OBJECT

public:
    enum Roles {
        NameRole = Qt::UserRole + 0,          // QString
        NotaryIDRole = Qt::UserRole + 1,      // QString
        NotaryNameRole = Qt::UserRole + 2,    // QString
        UnitRole = Qt::UserRole + 3,          // int (identity::wot::claim::)
        UnitNameRole = Qt::UserRole + 4,      // QString
        AccountIDRole = Qt::UserRole + 5,     // QString
        BalanceRole = Qt::UserRole + 6,       // QString
        PolarityRole = Qt::UserRole + 7,      // int (-1, 0, or 1)
        AccountTypeRole = Qt::UserRole + 8,   // int (opentxs::AccountType)
        ContractIdRole = Qt::UserRole + 9,    // QString
    };
    enum Columns {
        NotaryNameColumn = 0,
        DisplayUnitColumn = 1,
        AccountNameColumn = 2,
        DisplayBalanceColumn = 3,
    };
```



*This screenshot is only an example.

AccountActivity

"metier/deps/opentxs/include/opentxs/interface/qt/AccountActivity.hpp"

- See: opentxs::ui::AccountActivityQt

Each row in the AccountActivity represents a different activity item for that account. (Such as an incoming Bitcoin transfer, an outgoing Ether payment, or an outgoing cheque).

To access the account activity, call:

```
opentxs::ui::AccountActivityQt* activity = identity_manager->getAccountActivity(accountId);
```

Since Qt uses pointers, be sure to check the return value of the above function for null:

```
if (null != activity) { … }
```

Qt properties available on the AccountActivity model:

```
class OPENTXS_EXPORT opentxs::ui::AccountActivityQt final : public qt::Model
{
    Q_OBJECT
    Q_PROPERTY(QObject* amountValidator READ getAmountValidator CONSTANT)
    Q_PROPERTY(QObject* destValidator READ getDestValidator CONSTANT)
    Q_PROPERTY(QObject* scaleModel READ getScaleModel CONSTANT)
    Q_PROPERTY(QString accountID READ accountID CONSTANT)
    Q_PROPERTY(
        int balancePolarity READ balancePolarity NOTIFY balancePolarityChanged)
    Q_PROPERTY(QVariantList depositChains READ depositChains CONSTANT)
    Q_PROPERTY(QString displayBalance READ displayBalance NOTIFY balanceChanged)
    Q_PROPERTY(
        double syncPercentage READ syncPercentage NOTIFY syncPercentageUpdated)
    Q_PROPERTY(
        QVariantList syncProgress READ syncProgress NOTIFY syncProgressUpdated)
```

The Qt columns and Qt roles are provided below, showing the data available for each row -- with each row representing a single activity item from the account's transaction history:

```
public:
    enum Roles {
        AmountRole = Qt::UserRole + 0,          // QString
        TextRole = Qt::UserRole + 1,            // QString
        MemoRole = Qt::UserRole + 2,            // QString
        TimeRole = Qt::UserRole + 3,            // QDateTime
        UUIDRole = Qt::UserRole + 4,            // QString
        PolarityRole = Qt::UserRole + 5,        // int, -1, 0, or 1
        ContactsRole = Qt::UserRole + 6,        // QStringList
        WorkflowRole = Qt::UserRole + 7,        // QString
        TypeRole = Qt::UserRole + 8,            // int, opentxs::StorageBox)
        ConfirmationsRole = Qt::UserRole + 9,   // int
    };
    enum Columns {
        TimeColumn = 0,
        TextColumn = 1,
        AmountColumn = 2,
        UUIDColumn = 3,
        MemoColumn = 4,
        ConfirmationsColumn = 5,
    };
```

Several Qt signals are made available so that the wallet UI can connect a Qt slot and react to these signals:

```
signals:
    void balanceChanged(QString) const;
    void balancePolarityChanged(int) const;
    void transactionSendResult(int, int, QString) const;
    void syncPercentageUpdated(double) const;
    void syncProgressUpdated(int, int) const;
```

Public methods available on AccountActivity:

```
public:
    // NOLINTNEXTLINE(modernize-use-trailing-return-type)
    Q_INVOKABLE int sendToAddress(
        const QString& address,
        const QString& amount,
        const QString& memo,
        int scale = 0) const noexcept;
    // NOLINTNEXTLINE(modernize-use-trailing-return-type)
    Q_INVOKABLE int sendToContact(
        const QString& contactID,
        const QString& amount,
        const QString& memo,
        int scale = 0) const noexcept;
    // NOLINTNEXTLINE(modernize-use-trailing-return-type)
    Q_INVOKABLE QString getDepositAddress(const int chain = 0) const noexcept;
    // NOLINTNEXTLINE(modernize-use-trailing-return-type)
    Q_INVOKABLE bool validateAddress(const QString& address) const noexcept;
    // NOLINTNEXTLINE(modernize-use-trailing-return-type)
    Q_INVOKABLE QString validateAmount(const QString& amount) const noexcept;

    QString accountID() const noexcept;
    int balancePolarity() const noexcept;
    // Each item in the list is an opentxs::blockchain::Type enum value cast to
    // an int
    QVariantList depositChains() const noexcept;
    QString displayBalance() const noexcept;
    AmountValidator* getAmountValidator() const noexcept;
    DestinationValidator* getDestValidator() const noexcept;
    DisplayScaleQt* getScaleModel() const noexcept;
    auto headerData(
        int section,
        Qt::Orientation orientation,
        int role = Qt::DisplayRole) const noexcept -> QVariant final;
    double syncPercentage() const noexcept;
    QVariantList syncProgress() const noexcept;
```

The public methods *sendToAddress()* and *sendToContact()* allow the wallet developer to easily send funds from this account.

getDepositAddress() returns the next receiving address for the account.

validateAddress(address) returns success/failure whether or not the address (from user input) is in the right format that it can be used as a recipient for a transfer.

validateAmount(amount) takes user input (amount string) processes it into a proper Amount based on the unit type, and then returns that amount as a QString formatted for display in the UI.

displayBalance() returns the balance of the account, as a string formatted for display in the UI.

syncPercentage() indicates the synchronization progress of the account as a percentage from 0 to 100.

balancePolarity() indicates whether the account balance is a positive or negative amount.

ContactList

"metier/deps/opentxs/include/opentxs/interface/qt/ContactList.hpp"

- See: opentxs::ui::ContactListQt

Each row in the ContactList represents a different contact in the wallet user's contact list. The columns and Qt roles are provided below, showing the data available for each row. (For each contact).

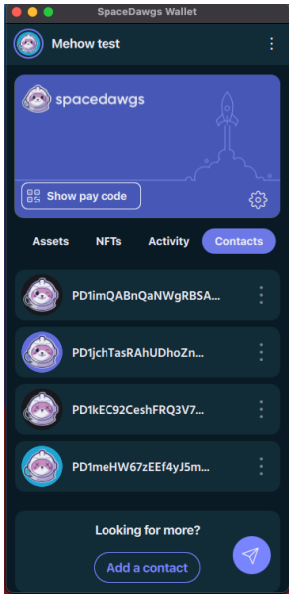
To access the contact list, call:

```
opentxs::ui::ContactListQt* contacts = identity_manager->getContactList();
```

Since Qt uses pointers, be sure to check the return value of the above function for null:

```
if (null != contacts) { … }
```

```
class OPENTXS_EXPORT opentxs::ui::ContactListQt final : public qt::Model
{
    Q_OBJECT

public:
    // NOLINTNEXTLINE(modernize-use-trailing-return-type)
    Q_INVOKABLE QString addContact(
        const QString& label,
        const QString& paymentCode = "",
        const QString& nymID = "") const noexcept;

public:
    enum Roles {
        IDRole = Qt::UserRole + 0,       // QString
        NameRole = Qt::UserRole + 1,     // QString
        ImageRole = Qt::UserRole + 2,    // QPixmap
        SectionRole = Qt::UserRole + 3,  // QString
    };
    // This model is designed to be used in a list view
    enum Columns {
        NameColumn = 0,
    };
```



*This screenshot is only an example.

addContact(), seen above, provides an easy method for adding new contacts to the wallet. The label parameter is for passing the display name for the new contact. The other two parameters are optional, but usually a payment code should be provided.

Note: In the above screenshot, in the Name column for each contact, a payment code is being displayed as the display name for each contact. However, if an actual display name is set, then that display name will appear there instead of the payment code, which is only shown when the normal display name is empty or otherwise unavailable. (For example, perhaps the display name is inside the credentials for a new contact, and OT is still downloading those new credentials, and so the display name is not yet available).

ActivityThread

"metier/deps/opentxs/include/opentxs/interface/qt/ActivityThread.hpp"

- See: opentxs::ui::ActivityThreadQt

Each row in the ActivityThread represents a different activity item between the wallet user and one of his contacts. An activity item may be (for example) an incoming Bitcoin transfer, an outgoing Ether payment, an outgoing cheque, **or an incoming or outgoing chat message.** All of the activity items in a given activity thread are between the wallet user and one of his contacts. (For example, all of the history between you and Alice, or all of the history between you and Bob).

To access the activity thread, call:

```
opentxs::ui::ActivityThreadQt* activity = identity_manager->getActivityThread(contactId);
```

Since Qt uses pointers, be sure to check the return value of the above function for null:

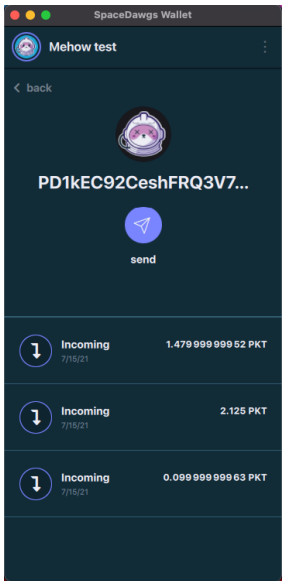
```
if (null != activity) { … }
```

Qt properties available on the ActivityThread model:

```
class OPENTXS_EXPORT opentxs::ui::ActivityThreadQt final : public qt::Model
{
    Q_OBJECT
    Q_PROPERTY(bool canMessage READ canMessage NOTIFY canMessageUpdate)
    Q_PROPERTY(QString displayName READ displayName NOTIFY displayNameUpdate)
    Q_PROPERTY(QString draft READ draft WRITE setDraft NOTIFY draftUpdate)
    Q_PROPERTY(QObject* draftValidator READ draftValidator CONSTANT)
    Q_PROPERTY(QString participants READ participants CONSTANT)
    Q_PROPERTY(QString threadID READ threadID CONSTANT)
```

The Qt columns and Qt roles are provided below, showing the data available for each row -- with each row representing a single activity item from the wallet user's history with a specific contact:

```
enum Roles {
    AmountRole = Qt::UserRole + 0,    // QString
    LoadingRole = Qt::UserRole + 1,   // bool
    MemoRole = Qt::UserRole + 2,      // QString
    PendingRole = Qt::UserRole + 3,   // bool
    PolarityRole = Qt::UserRole + 4,  // int, -1, 0, or 1
    TextRole = Qt::UserRole + 5,      // QString
    TimeRole = Qt::UserRole + 6,      // QDateTime
    TypeRole = Qt::UserRole + 7,      // int, opentxs::StorageBox
    OutgoingRole = Qt::UserRole + 8,  // bool
    FromRole = Qt::UserRole + 9,      // QString
};
enum Columns {
    TimeColumn = 0,
    FromColumn = 1,
    TextColumn = 2,
    AmountColumn = 3,
    MemoColumn = 4,
    LoadingColumn = 5,
    PendingColumn = 6,
};
```



*This screenshot is only an example.

Several Qt signals are made available so that the wallet UI can connect a Qt slot and react to these signals:

```
signals:
    void canMessageUpdate(bool) const;
    void displayNameUpdate() const;
    void draftUpdate() const;
```

Public methods available on ActivityThread:

```
public:
    // NOLINTNEXTLINE(modernize-use-trailing-return-type)
    Q_INVOKABLE bool pay(
        const QString& amount,
        const QString& sourceAccount,
        const QString& memo = "") const noexcept;
    // NOLINTNEXTLINE(modernize-use-trailing-return-type)
    Q_INVOKABLE QString paymentCode(const int currency) const noexcept;
    // NOLINTNEXTLINE(modernize-use-trailing-return-type)
    Q_INVOKABLE bool sendDraft() const noexcept;
```

```
auto canMessage() const noexcept -> bool;
auto displayName() const noexcept -> QString;
auto draft() const noexcept -> QString;
auto draftValidator() const noexcept -> QValidator*;
auto headerData(
    int section,
    Qt::Orientation orientation,
    int role = Qt::DisplayRole) const noexcept -> QVariant final;
auto participants() const noexcept -> QString;
auto threadID() const noexcept -> QString;
```

displayName() returns the display name for this contact.

pay() provides an easy method for sending money to this contact.

paymentCode() returns the payment code for this contact as a QString.

canMesasge() returns a Boolean indicating whether or not this contact is message-able. It also triggers background processes in OT to obtain message-ability for a given contact. If updated information becomes available, or if the contact otherwise subsequently becomes message-able, the *canMessageUpdate()* signal will be emitted, allowing the UI to react to that event.

Changes to the *draft* property will cause this model to emit a *draftUpdate()* signal. At some point the “Send” button in the UI will be clicked by the user, triggering the *sendDraft()* method and causing the current draft chat message to actually be sent to the contact.