Chapter 8

Personal Information Management APIs

As you start developing Cascades business and productivity apps, you will realize the necessity for leveraging core services such as searching contacts, sending messages, and managing calendar entries. The aforementioned services fall under the personal information management (PIM) umbrella and refer to the tools used to manage the user's personal and professional lives. One approach would be to implement the PIM services in your own application, which would quickly become daunting. Also from a user perspective, providing functionality already covered by the core applications would be less than ideal. A better approach would therefore be to reuse the preexisting PIM services provided by the BlackBerry 10 core applications and leverage them in your own apps. You can essentially achieve this in two ways:

- *Use service APIs*: All BlackBerry 10 PIM applications provide an API for interfacing with their data stores. To leverage the APIs, you will have to link your application against the `bbpim` library and use service classes to access the PIM functionality.
- *Use the invocation framework*: Use this to invoke core applications from your own app.

I will cover the PIM service APIs in this chapter. The invocation framework will be the subject of Chapter 10. After having read this chapter, you will

- Understand how user accounts are linked to service providers on the BlackBerry 10 device.
- Have a good overview of the APIs used for interfacing to the BlackBerry 10 PIM applications.

Personal Information Management

In a broad sense, "personal information management" refers to the tools used by the user to organize his personal and professional lives. The following are the corresponding BlackBerry 10 core applications:

- *Contacts*: Enables the user to manage his contacts and store relevant information such as a picture, work number, mobile number, e-mail, and so forth.
- *Calendar*: Enables the user to manage meetings, appointments, and events.
- *Messaging*: Enables the user to send and receive e-mail and short text messages.
- *Notebooks*: Provides a productivity app for collecting, managing, and organizing information that the user wants to remember. Information is organized in folders.

In this chapter, I will cover the Contacts, Calendar, and Messaging APIs, which correspond to the PIM services used most often. You can also use this chapter as a reference for the PIM APIs.

PIM APIs

This section describes the APIs used for accessing the PIM applications described in the previous section. You will see that the APIs always provide a service class, which corresponds to the API's interface to the target application's database. The material will be presented in a top-down approach by always starting with the service interface, and then explaining the remaining classes used in calling the interface.

> **Note** To use the PIM APIs, you will have to add `LIBS += -lbbpim` to your application's `.pro` file.

Service Types

The PIM APIs define service types, which correspond to broad categories of services such as messaging, calendars, contacts, geolocation, phone, and so on. The `Service` class encapsulates this information in the `Service::Type` enumeration (the values corresponding to PIM services are as follows):

- `Service::Calendars`: Represents a calendar service type. A calendar service can be used to manage meetings and appointments.
- `Service::Contacts`: Represents a contacts service type. A contact service can be used for managing user contacts, including data such as e-mail, phone numbers, and so forth.
- `Service::Messages`: Represents a message service type. A message service can be used for sending and receiving messages. A message could be an e-mail message, a short text message, or even a tweet.
- `Service::NoteBook`: Represents a notebook service type, which contains a list of items. A notebook could be something as simple as a grocery list.

As you will see in the next section, the actual services are implemented by *service providers*, which are linked to accounts on the device (for example, the `caldav` service provider can be used for accessing calendar services).

Service Providers

A service provider typically implements a service type. Note that a given service type can be implemented by multiple service providers, which in turn can correspond to multiple accounts on the device (for example, the calendar service is implemented by the `localcalendar` provider, which corresponds to the device's "local" calendar account, and the `caldav` service provider, which could be linked to a Google calendar account). In C++, you can use the `QList<Provider> AccountService::providers()` method call to retrieve the list of all service providers available on the device. You can then determine additional information about a service provider using the `Provider` class:

- `QString Provider::id()`: Returns this provider's id. Typical examples of provider ids are `localcalendar`, `localcontacts`, `sms-mms`, `facebook`, `caldav`, `imapemail`, and so forth.
- `QString Provider::name()`: Returns this provider's name. You can use the name property to display a user-friendly string to the user.
- `bool Provider::isServiceSupported(Service::Type service)`: Returns whether or not the service type is supported by the provider.
- `bool Provider::isSocial()`: Returns whether this service provider is a social networking service.
- `bool Property::EnterpriseType Provider::isEnterprise()`: Returns whether or not this service provider is an enterprise service. Possible values for `EnterpriseType` are `EnterpriseUnknown`, `NonEnterprise`, and `Enterprise`.
- `QList<QString> Provider::settingsKeys()`: Returns this provider's settings keys. You can consider the settings keys as a generic way of specifying the parameters required for creating a new account linked to the corresponding service provider. In other words, each provider will define its own set of keys that you will have to use when linking an account to the provider.
- `QVariant Provider::settingsProperty(const QString& key, Property::Field property)`: Returns metainformation for the given settings key. For example, you can use this method to determine the type of a given key using `Property::Type` as the second parameter. The returned `QVariant` will contain a `string` describing the type. The possible values are `number`, `boolean`, `string`, and `email`).

Accounts

An `Account` object represents a user account stored on the device. Using the `Account` class, you can retrieve information such as the account's id, and most importantly, to which provider the account is linked. Important `Account` methods are summarized as follows (the next section will show you how to retrieve user accounts stored on the device):

- `Account(const Provider& provider)`: Instantiates a new account object linked to the given provider. All the account properties are set to the default values as defined by the provider.
- `AccountKey Account::id()`: Returns this account's ID. Note that you will need the `AccountKey` to use service classes such as the `CalendarService` and `MessageService`.
- `Provider Account::provider()`: Returns the provider associated to this account.
- `void Account::setSettingsValue(const QString& key, const QVariant& value)`: Assigns `value` to the corresponding `key`. The `key` is defined by the provider linked to this account (also see `Provider::settingsKeys()`).

AccountService Class

You can use the `AccountService` class to determine the service providers registered on the user's device, as well as the corresponding accounts. The following list reviews important `AccountService` methods:

- `Result AccountService::createAccount(const QString& providerId, Account& accountData)`: Creates a new account linked to the service provider given by `providerId`.
- `QList<Account> AccountService::accounts()`: Retrieves the list of all accounts stored on the device.
- `QList<Account> AccountService::accounts(Service::Type service, const QString& providerId)`: Retrieves the list of accounts stored on the device for a given service type and provider. The `providerId` string is given by `Provider::id()` (see the description in the "Service Providers" section).
- `Account AccountService::defaultAccount(Service::Type type)`: Returns the default account for a given service type. The `Service::Type` enumeration can take the following values: `Calendars`, `Contacts`, `Notebook`, `Geolocations`, `Linking`, `Memos`, `Messages`, `Tags`, `Tasks`, and `Phone`.
- `QMap<Service::Type, Account> AccountService::defaultAccounts()`: Returns a map of default accounts by service type.
- `QList<Provider> AccountService::providers()`: Retrieves the list of all provider objects.
- `QList<Account> AccountService::accounts(Service::Type service)`: Retrieves the list of `Account` objects currently synchronizing data for the given service type.

Creating a New Account

You can use the `AccountService` class to create a new account linked to a given service provider by performing the following steps:

1. Retrieve the provider's keys, which correspond to the account parameters that you will have to set.
2. Instantiate an `Account` object by passing the provider object to the `Account` object's constructor. Update the `Account` object using the provider keys.
3. Create the actual account using the `AccountService::createAccount(const QString& providerId, Account)` method.

Listing 8-1 outlines the process in practice (note that the `getKeyValue()` method, which is used to retrieve a key value, is not shown. In practice, the key values could be provided by a user-entered QML form or loaded using app settings at application start-up).

Listing 8-1. Account Creation

```
const QString providerId = "imapemail";
const Provider provider = m_accountService->provider(providerId);

Account account(provider);

// Iterate over all of the provider's settings keys
foreach (const QString &key, provider.settingsKeys()) {
    QVariant value  = getKeyValue(key);
    account.setSettingsValue(key, value);
}

m_accountService->createAccount(provider.id(), account);
```

Searching for Accounts

As illustrated in Listing 8-2, you can use the `AccountService` class to search accounts linked to a given provider.

Listing 8-2. Account Creation

```
#include <bb/pim/account/AccountService>

using namespace bb::pim::account;

AccountService accountService;
QList<Account> accounts = accountService.accounts(Service::Messages,"emailemap");
for (int i = 0; i < accounts.size(); i++) {
    cout << "display name: " + accounts[i].displayName().toStdString() << endl;
}
```

In a similar way, if you wanted to retrieve the accounts linked to the caldav provider, you could use the following method call: `accountService.accounts(Service::Calendar, "caldav")`

In practice, as you will see in the following sections, you will need the `Account` ID to update the corresponding PIM app.

Contacts API

You can use the Contacts API to create, update, and delete contacts stored on the device. Typically, when you add a new contact, you can set the contact's attributes such as e-mails, postal addresses, phone numbers, pictures, and so on. Using the `ContactService` class, the following sections will illustrate basic operations of the Contacts database.

> **Note** To access the Contacts database, you need to add the `access_pimdomain_contacts` permission in your project's `bar-descriptor.xml` file.

ContactService

As with accounts and the `AccountService` class, the `ContactService` class is the central interface for manipulating contacts stored on the device. The following summarizes `ContactService` methods:

- `Contact ContactService::createContact(const Contact& contact, bool isWork)`: Creates a new contact and adds it to the Contacts database. If `isWork` is `true`, the contact will be created in the enterprise perimeter; otherwise, the contact will be created in the personal perimeter.
- `Contact ContactService::contactDetails(ContactId id)`: Retrieves the full details of the contact given by id.
- `ContactService::updateContact(const Contact& contact)`: Updates an existing contact. Note that you need to be sure that you have retrieved the contact using `ContactService::contactDetails(ContactId id)`. Only contacts retrieved with the previous method return the full contact data. Other methods return partial contact information and the call to `ContactService::updateContact(const Contact& contact)` might then overwrite the database with incomplete data.
- `QList<Contact> ContactService::searchContacts(const ContactSearchFilters& filters)`: Retrieves a list of contacts based on the given search filter. The default search fields are first name, last name, company name, phone, and e-mail.
- `QList<Contact> ContactService::contacts(const ContactListFilters& filters)`: Retrieves a list of contacts based on the given list filters.
- `void ContactService::deleteContact(ContactId contactId)`: Deletes the contact whose `ContactId` is id.

Creating a New Contact

Listing 8-3 shows you how to create a new contact in the Contacts database.

Listing 8-3. Creating a New Contact

```
#include <bb/pim/contacts/ContactService>
#include <bb/pim/contacts/Contact>
#include <bb/pim/contacts/ContactAttributeBuilder>
#include <bb/pim/contacts/ContactBuilder>

using namespace bb::pim::contacts;

ContactService contactService;

QString firstName("Anwar");
QString lastName("Ludin");
QDateTime birthday(QDate(1973, 1, 21));
QString email("anwar@aludin.com");

ContactBuilder builder;

// Set the first name
builder.addAttribute(ContactAttributeBuilder()
                     .setKind(AttributeKind::Name)
                     .setSubKind(AttributeSubKind::NameGiven)
                     .setValue(firstName));

// Set the last name
builder.addAttribute(ContactAttributeBuilder()
                     .setKind(AttributeKind::Name)
                     .setSubKind(AttributeSubKind::NameSurname)
                     .setValue(lastName));

// Set the birthday
builder.addAttribute(ContactAttributeBuilder()
                     .setKind(AttributeKind::Date)
                     .setSubKind(AttributeSubKind::DateBirthday)
                     .setValue(birthday));

// Set the email address
builder.addAttribute(ContactAttributeBuilder()
                     .setKind(AttributeKind::Email)
                     .setSubKind(AttributeSubKind::Work)
                     .setValue(email));

// Set the postal address
builder.addPostalAddress(ContactPostalAddressBuilder().setCity("Geneva")
                         .setCountry("Switzerland")
                         .setLine1("2 rue de la Muse")
                         .setPostalCode("1205")
                         .setSubKind(AttributeSubKind::Work));
```

```
// Set photo
builder.addPhoto(ContactPhotoBuilder()
                 .setOriginalPhoto("/accounts/1000/shared/photos/aludin.jpg"));

// Save the contact to persistent storage
contactService.createContact(builder, false);
```

The code is relatively straightforward. The easiest way to create a new contact is to use a `ContactBuilder` instance. You can also assign attributes to the contact using a `ContactAttributeBuilder` instance (as illustrated in Listing 8-2, you can specify the attribute's kind, subkind, and value). For adding a postal address, you should use a `ContactPostalAddressBuilder`. You can also assign a photo to the contact using a `ContactPhotoBuilder`. Finally, once the contact's attributes have been set, you can call the `ContactService::createContact(Contact contact, bool isWork)` method to add the new contact to the Contacts database (note that you can pass the `ContactBuilder` instance directly to the `ContactService::createContact()` method because it provides a conversion operator, which will create a `Contact` object from the `ContactBuilder` object).

> **Note** To access the contact's photo in a shared folder on the file system, you must add the Shared Files permission to your project's `bar-descriptor.xml` file.

And finally, Figure 8-1 illustrates the newly created contact displayed in the BlackBerry 10 Contacts app.

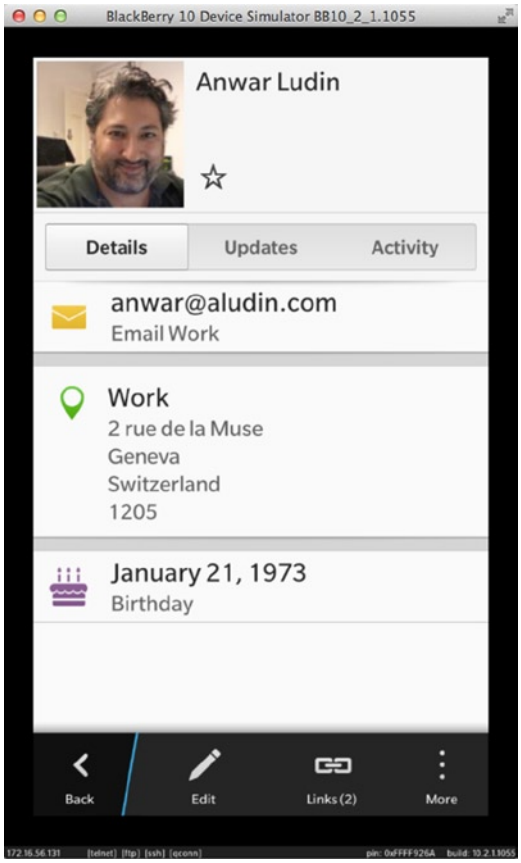

Figure 8-1. Newly created contact

Updating a Contact

You can also update an existing contact using the `ContactService::updateContact()` method, as illustrated in Listing 8-4.

Listing 8-4. Updating a Contact

```
#include <bb/pim/contacts/ContactService>
#include <bb/pim/contacts/Contact>
#include <bb/pim/contacts/ContactAttributeBuilder>
#include <bb/pim/contacts/ContactBuilder>

ContactService contactService

int ContactId = 100;    // alternatively use a search to get the contact

Contact contact = contactService->contactDetails(contactId);
if (contact.id()) {
    // Create a builder to modify the contact
    ContactBuilder builder = contact.edit();
```

```
    // Update the single attributes
    updateContactAttribute<QString>(builder, contact,
                                    AttributeKind::Name, AttributeSubKind::NameGiven,
                                    "Jack");
    updateContactAttribute<QString>(builder, contact,
                                    AttributeKind::Name, AttributeSubKind::NameSurname,
                                    "Smith");
    updateContactAttribute<QDateTime>(builder, contact,
                                      AttributeKind::Date, AttributeSubKind::DateBirthday,
                                      QDateTime(QDate(1980,3,21)));
    updateContactAttribute<QString>(builder, contact,
                                    AttributeKind::Email, AttributeSubKind::Other, "jsmith@aludin.com");

    // Save the updated contact back to persistent storage
    contactService->updateContact(builder);
}
```

As shown in Listing 8-4, you need to first retrieve the contact's full details using the `ContactService::ContactDetails(ContactId id)` method before updating the contact. You can then use the `ContactBuilder` returned by the `Contact::edit()` method to update the contact's attributes (the code uses the templated `updateContactAttribute<T>()` helper function to update the contact's attributes (see Listing 8-5).

Listing 8-5. Updating Contact Attributes

```
template<typename T>
static void updateContactAttribute(ContactBuilder &builder,

const Contact &contact, AttributeKind::Type kind,
    AttributeSubKind::Type subKind, const T &value)
{
    // Delete previous instance of the attribute
    QList<ContactAttribute> attributes = contact.filteredAttributes(kind);
    foreach (const ContactAttribute &attribute, attributes)
    {
        if (attribute.subKind() == subKind)
            builder.deleteAttribute(attribute);
    }

    // Add new instance of the attribute with new value
    builder.addAttribute(ContactAttributeBuilder().setKind(kind)
                         .setSubKind(subKind).setValue(value));
}
```

Note how the code first deletes all previous instances of the attribute in the contact's entry, and then updates the builder to include the new attribute value.

Searching for Contacts

Besides creating and updating contacts, you can also use the `ContactService` class to search for contacts by matching search criteria. There are two ways to perform a search. First, you can create a `ContactSearchFilters` instance that you pass to the `ContactService::searchContacts(const ContactSearchFilters& filter)` method. In this case, you must at least specify a search value, which is a string, but you can also further refine the search criteria by specifying search fields using the `SearchField::Type` enumeration (if you don't specify any search fields, the default `first name`, `company name`, `phone`, and `email` fields will be used for matching the search value). Besides search fields, you can also specify whether an attribute is present or not in the contact's entry.

Alternatively, you can use a `ContactListFilters` instance and pass it to the `ContactService::contacts(const ContactListFilters& filter)` method. In both cases, you can control the number of returned search results by using the `ContactSearchFilters::setLimit()` and the `ContactListFilters::setLimit()` methods (if you don't specify a search limit, 20 values will be returned at most; note that you can also choose to retrieve all the results corresponding to a search by setting the limit to 0).

The following summarizes important `ContactSearchFilters` methods (for a detailed description of `ContactSearchFilters` and `ContactListFilters`, consult BlackBerry's online documentation):

- `ContactSearchFilters& ContactSearchFilters::setSearchValue(const QString& value)`: Sets the string to search in the list of contacts.
- `ContactSearchFilters& ContactSearchFilters::setSearchFields(const QList<SearchField::Type>& fields)`: Sets the search fields that the search applies to. These fields are searched for the value set by the previous method.
- `ContactSearchFilters& ContactSearchFilters::setHasAttribute(AttributeKind::Type present)`: Filters the search results to contain only contacts with the provided attribute kind.
- `ContactSearchFilter& ContactSearchFilter::setShowAttributes(bool value)`: Specifies whether or not to include attributes in the search results. If `true`, attributes are returned. If `true` along with `ContactSearchFilter::setHasAttribute()`, then only the matching attributes are returned.
- `ContactSearchFilters& ContactSearchFilters::setLimit(int limit)`: Sets the maximum number of results returned by the search.
- `ContactSearchFilters& ContactSearchFilters::setAnchorId(ContactId anchor, bool inclusive)`: Sets the current `anchor` for paging. If `inclusive` is `true`, `anchor` is included in the search results; otherwise, the contact after `anchor` is returned in the search results (see the next section about paging).

The code shown in Listing 8-6 illustrates how to perform a search in practice (the code is adapted from the BlackBerry 10 address book sample app and is used to update a `ListView` data model with the search results).

Listing 8-6. AddressBook::filterContacts()

```
void AddressBook::filterContacts()
{
    QList<Contact> contacts;

    if (m_filter.isEmpty()) {
        // No filter has been specified, so just list all contacts
        ContactListFilters filter;
        filter.setLimit(0)
        contacts = m_contactService->contacts(filter);
    } else {
        // Use the entered filter string as search value
        ContactSearchFilters filter;
        filter.setSearchValue(m_filter);

        contacts = m_contactService->searchContacts(filter);
    }

    // Clear the old contact information from the model
    m_model->clear();

    // Iterate over the list of contact IDs
    foreach (const Contact &idContact, contacts) {
        // Fetch the complete details for this contact ID
        const Contact contact = m_contactService->contactDetails(idContact.id());

        // Copy the data into a model entry
        QVariantMap entry;
        entry["contactId"] = contact.id();
        entry["firstName"] = contact.firstName();
        entry["lastName"] = contact.lastName();

        const QList<ContactAttribute> emails = contact.emails();
        if (!emails.isEmpty())
            entry["email"] = emails.first().value();

        // Add the entry to the model
        m_model->insert(entry);
    }
}
```

In the previous example, if the filter string given by `m_filter` is empty, the code simply retrieves all contacts using the `ContactService::contacts()` method. Otherwise, a `ContactSearchFilter` instance is created with the search criteria and passed to the `ContactService::searchContacts()` method. Finally, the `ContactService::contactDetails()` method is used to retrieve a given contact's full attributes (as mentioned previously, the search results will only return a partial list of attributes; if you need the full list of attributes, you must call `ContactService::contactDetails()`).

Paging

You can use paging to navigate through a partial list of contacts. In practice, paging is important for performance reasons because it avoids the search to block the UI thread (as a good rule of thumb, if your search criteria returns more than 200 values, you should consider paging). Listing 8-7 illustrates how to use paging in practice.

Listing 8-7. Paging

```
ContactSearchFilters filter;
filter.setSearchValue("Anwar");
filter.setLimit(20);
QList<Contact> contactPage;
do
{
    contacts = service.searchContacts(filter);
    process(contactPage);
    if (contactPage.size() == maxLimit)
    {
        filter.setAnchorId(contactPage[maxLimit-1].id());
    }
    else
    {
        break;
    }
} while (true);
```

The previous code uses a `do-while` loop to process search results in pages of size 20. Note that you need to update during an iteration the `anchor id`, which corresponds to the last element returned by the previous page, in order to move to the next logical page. Finally, you know that you are processing the last page when the current page size is less than the maximum page limit. At this point, you need to break out of the loop.

Asynchronous Search

An alternative to paging is to use an asynchronous search to avoid blocking the main UI thread (the golden rule for building enticing Cascades apps is a nonblocking responsive UI). As mentioned in Chapter 3, to perform an asynchronous operation, you need to create a `worker` object and start it in a separate thread from the main UI thread.

To illustrate how you can perform an asynchronous search in practice, Listing 8-8 gives you the `AsynchSearch` class definition.

Listing 8-8. Asynchronous Search

```
#include <QObject>
#include <QString>
#include <QList>

#include <bb/pim/contacts/ContactService>
#include <bb/pim/contacts/Contact>
#include <bb/pim/contacts/ContactSearchFilters>
```

```
using namespace bb::pim::contacts;

class AsynchSearch: public QObject {
    Q_OBJECT
public:
    AsynchSearch(QObject* parent = 0) : QObject(parent) {};
    virtual ~AsynchSearch() {};
public slots:
    void doSearch();
public:
    void setFilter(QString filter) {
        m_filter = filter;
    }
    QString filter() {
        return m_filter;
    }

signals:
    void searchFinished(QList<Contact>);

private:
    QString m_filter;
    ContactService m_contactService;
};
```

> **Note** You can download a modified version of the AddressBook sample app using asynchronous searches from this book's GitHub repository at https://github.com/aludin/BB10Apress.

As illustrated in the `AsynchSearch` class definition, the class returns its search results using the `searchFinished(QList<Contact> contacts)` signal. The actual search is performed in the `AsynchSearch::doSearch()` method shown in Listing 8-9.

Listing 8-9. AsynchSearch::doSearch()

```
#include "AsynchSearch.h"

void AsynchSearch::doSearch() {
    QList<Contact> contacts;
    QList<Contact> contactsDetails;
    if (m_filter.isEmpty()) {
        // No filter has been specified, so just list all contacts
        ContactListFilters filter;
        filter.setLimit(0);
        contacts = m_contactService.contacts(filter);
        foreach (Contact c, contacts)
```

```
        {
            // Fetch the complete details for this contact ID
            const Contact contact = m_contactService.contactDetails(c.id());
            contactsDetails.append(contact);
        }
        emit searchFinished(contactsDetails);
    } else {
        // Use the entered filter string as search value
        ContactSearchFilters filter;
        filter.setSearchValue(m_filter);
        contacts = m_contactService.searchContacts(filter);
        foreach (Contact c, contacts)
        {
            // Fetch the complete details for this contact ID
            const Contact contact = m_contactService.contactDetails(c.id());
            contactsDetails.append(contact);
        }
        emit searchFinished(contactsDetails);
    }
}
```

Here again, the code uses the `m_filter` variable to retrieve the search results in a similar way to Listing 8-6 (the main difference comes from the fact that the contact details are not used to update a data model). Finally, as mentioned, when the search has completed, the `searchFinished()` signal is emitted with the list of contacts corresponding to the search criteria. The updated version of `AddressBook::filterContacts()`, which performs an asynchronous search, is given in Listing 8-10.

Listing 8-10. AddressBook::filterContacts(), Updated

```
void AddressBook::filterContacts() {
    QThread* thread = new QThread;
    AsynchSearch* asynch = new AsynchSearch;
    asynch->setFilter(m_filter);
    asynch->moveToThread(thread);

    bool result = connect(thread, SIGNAL(started()), asynch, SLOT(doSearch()));
    Q_ASSERT(result);
    result = connect(asynch, SIGNAL(searchFinished(QList<Contact>)), this,
                     SLOT(onSearchFinished(QList<Contact>)));
    Q_ASSERT(result);

    result = connect(asynch, SIGNAL(searchFinished(const QList<Contact>)),
                     thread, SLOT(quit()));
    Q_ASSERT(result);
    result = connect(asynch, SIGNAL(searchFinished(const QList<Contact>)),
                     asynch, SLOT(deleteLater()));
    Q_ASSERT(result);
    result = connect(thread, SIGNAL(finished()), thread, SLOT(deleteLater()));
    Q_ASSERT(result);

    thread->start();
}
```

As illustrated in Listing 8-10, the updated version of `AddressBook::filterContacts()` creates a new `Thread` and initializes an `AsynchSearch` object so that it will be run in the separate `Thread` by moving the `AsynchSearch` instance to the new thread context.

The signals and slot connections are configured as follows:

- The `QThread::started()` signal is connected to the `AsynchSearch::doSearch()` slot to perform the search when the thread is started.
- The `AsynchSearch::searchFinished()` signal is connected to the `AddressBook::onSearchCompleted()` slot to return the search results to the main UI thread.
- The same `AsynchSearch::searchFinished()` signal is also connected to the secondary thread's `QThread::quit()` slot, which will in turn emit the `QThread::finished()` signal.
- Memory management and cleanup is handled by the `AsynchSearch::searchFinished()` and `QThread::finished()` signals, which call their corresponding `deleteLater()` slots.

Finally, the `AddressBook::onSearchCompleted()` slot, which is used to update the data model, is shown in Listing 8-11.

Listing 8-11. AddressBook::onSearchFinished()

```
void AddressBook::onSearchFinished(QList<Contact> contacts) {

    // Clear the old contact information from the model
    m_model->clear();

    // Iterate over the list of contact IDs
    foreach (Contact c, contacts)
    {
        // Copy the data into a model entry
        QVariantMap entry;
        entry["contactId"] = c.id();
        entry["firstName"] = c.firstName();
        entry["lastName"] = c.lastName();

        const QList<ContactAttribute> emails = c.emails();
        if (!emails.isEmpty())
            entry["email"] = emails.first().value();
            // Add the entry to the model
        m_model->insert(entry);
    }
}
```

Also note that you need to register with the Qt type system the `QList<Contact>` type used as a parameter in the *interthread* signal (in interthread signals, slots are not called immediately. but at a "later stage" in the emitting thread's event loop; therefore, the parameters passed to a slot located in a different thread need to be saved and restored by the Qt type system); see Listing 8-12.

Listing 8-12. main.cpp

```
qRegisterMetaType< QList<Contact> >( "QList<Contact>" );
```

Using a ContactsPicker

You can include the `ContactsPicker` control in your app if you want to provide a search interface similar to the core Contacts app (the `ContactPicker` control uses a `Card` behind the scenes to display its UI; you will find out about `Card`s when we cover the invocation framework in Chapter 10). As you will see shortly, you can specify whether the `ContactsPicker` is configured in single-selection or multiselection mode. To use the `ContactsPicker` in QML, you must first register the corresponding C++ type with the QML type system (note that you must also register the `ContactSelectionMode` class, which is used for setting the selection mode; see Listing 8-13 and Listing 8-14).

Listing 8-13. main.cpp

```
qmlRegisterType<ContactPicker>("bb.cascades.pickers", 1, 0, "ContactPicker");

qmlRegisterUncreatableType<ContactSelectionMode>("bb.cascades.pickers", 1, 0,
    "ContactSelectionMode", "Can't instantiate enum");
```

And finally, Listing 8-14 shows you how to use the `ContactPicker` control in QML.

Listing 8-14. ContactPicker

```
import bb.cascades 1.2
import bb.cascades.pickers 1.0
Page {
    Container {
        Button {
            text: "Open contact picker"
            onClicked: {
                picker.open();
            }
        }
        Label {
            id: result
            text: "You chose contact: "
        }


    attachedObjects: [
            ContactPicker{
                id: picker
                mode: ContactSelectionMode.Multiple
                onContactsSelected:{
                    for(var i=0; i< contactIds.length; i++){
                        console.log(contactIds[i]);
                    }
                }
            }
        ]
    }
}
```

When the `ContactPicker` control is displayed, the user can select multiple contacts (see Figure 8-2). When the user completes his selection and touches the Done button, the `contactsSelected()` signal is emitted with a list of selected contact ids (if you don't want the user to be able to select multiple contacts, you can change the selection mode to `ContactSelectionMode.Single` and respond to the `contactSelected(id)` signal).

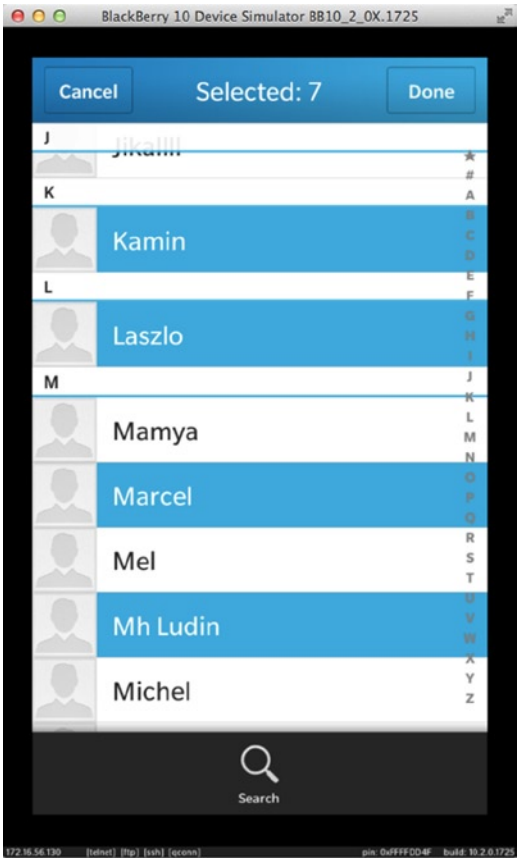

Figure 8-2. ContactPicker in multiselection mode

Calendar API

You can use the `CalendarService` class to add, update, and delete events in the Calendar database. Each event is represented by a `CalendarEvent` object, which should contain at least the following mandatory fields:

- Account ID: The account used for accessing the calendars. As mentioned previously, an account is linked to a service provider, which is either `localcalendar` or `caldav`.

- Folder ID: Each user account can in turn include multiple calendars identified by a folder ID. Therefore the "account ID, folder ID" pair uniquely identifies a user calendar on the device.
- Start time: The start time for this event (in C++ you can use a `QDateTime` object to specify this parameter; in QML you can use a `DatePicker`).
- End time: The end time for this event.
- Subject: The event's subject specified as a `QString`.

> **Note** To access the Calendar database, you need to set the `access_pimdomain_calendars` permission in your project's `bar-descriptor.xml` file.

CalendarService

The `CalendarService` class is the API entry point for accessing the Calendar database. You can use a `CalendarService` instance to manage calendars, events, attendees, and event locations. Note that all `CalendarService` methods provide a `Result::Type` parameter to indicate to the client application whether or not the API call was successful.

The following summarizes important `CalendarService` methods:

- `QList<CalendarFolder> CalendarService::folders(Result::Type* result)`: Returns all calendars folders from all calendar accounts (including remote calendars such as `caldav`; a `CalendarFolder` object's represents a distinct calendar).
- `QPair<AccountId, FolderId> CalendarService::defaultCalendar(Result::Type* result)`: Returns a pair of IDs that specify the default calendar (the default calendar is set by the user during device configuration. The setting is available using the Set Defaults action located under Settings ➤ Account Settings).
- `Result::Type CalendarService::createEvent(const CalendarEvent& event, const Notification& notification=0)`: Creates a new event in the Calendar database. You can optionally specify whether a notification should be sent to attendees.
- `QList<CalendarEvent> CalendarService::events(const EventSearchParameters& params, QResult::Type* result=0)`: Retrieves a list of events that match a specific search criteria identified by `params`. Note that depending on the search criteria, this method can potentially take a few seconds to complete. It would therefore be preferable not to call this method in the UI's main thread; use an asynchronous search instead.
- `Result::Type CalendarService::deleteEvent(const CalendarEvent& event, const Notification& notification)`: Deletes and removes an event from the Calendar database.

CalendarFolder

A `CalendarFolder` is a container for calendar events. You can use this class to determine calendar information such as name, type, owner e-mail address, and color (you can only update the calendar's color in the Calendar database).

CalendarEvent

A `CalendarEvent` object represents an event or meeting in the user's calendar. Apart from the mandatory fields discussed at the start of this section, you can add additional information to the event, including attendees, location, event details, whether the event is a birthday, and so on.

The following summarizes important `CalendarEvent` setters:

- `CalendarEvent::setAccountId(AccountId accountId)`: Sets the account ID for this CalendarEvent.
- `CalendarEvent::setFolderId(FolderId folderId)`: Sets the folder ID for this `CalendarEvent`.
- `CalendarEvent::setStartTime(const QDateTime& startTime)`: Sets the start time for this `CalendarEvent`.
- `CalendarEvent::setEndTime(const QDateTime& endTime)`: Sets the end time for this `CalendarEvent`.
- `CalendarEvent::setBody(const QString& body)`: Sets the body of this `CalendarEvent`. The body provides further details about the event.
- `CalendarEvent::setAllDay(bool allDay)`: Sets whether or not this `CalendarEvent` is an all-day event.
- `CalendarEvent::setAttendees(const QList<Attendee>& attendees)`: Sets the list of attendees for this `CalendarEvent`.
- `CalendarEvent::setLocation(const EventLocation& eventLocation)`: Sets the location for this `CalendarEvent`. `EventLocation`is a defined as a typedef `QString EventLocation`.

Attendee

An *attendee* is a participant to a meeting. You can use the `Attendee` class to specify information about the participant, such as his name, e-mail, and his role in the meeting (chair, required participant, optional participant, or nonparticipant included for information only).

The following summarizes important `Attendee` properties:

- `Attendee::setContactId(ContactId contactId)`: Sets the contact ID for this `Attendee`.
- `Attendee::setEmail(const QString& email)`: Sets the e-mail of this `Attendee`.

- `Attendee::setName(const QString& name)`: Sets the name of this `Attendee`.
- `Attendee::setRole(AttendeeRole::Type role)`: Sets the role of this `Attendee` (the possible values are `AttendeeRole::Invalid`, `AttendeeRole::Chair`, `AttendeeRole::ReqParticipant`, `AttendeeRole::OptParticipant`, and `AttendeeRole::NonParticipant`).

Creating a New Event

Putting all the pieces together, Listing 8-15 shows you how to create new events in the default calendar.

Listing 8-15. CalendarService

```
#include <bb/pim/calendar/CalendarService>
#include <bb/pim/calendar/CalendarEvent>
#include <bb/pim/calendar/CalendarFolder>

#include <bb/pim/calendar/Attendee>

using namespace bb::pim::calendar;

// Create the calendar service object
CalendarService calendarService;

// Create the calendar events
CalendarEvent firstEvent;

// Retrieve the IDs of the default calendar on the device
QPair<AccountId, FolderId> defaultCalendar = calendarService.defaultCalendarFolder();

// Specify information for the first event
firstEvent.setStartTime(QDateTime(QDate(2014, 03,11), QTime(10,00,00)));
firstEvent.setEndTime(QDateTime(QDate(2014, 03,11), QTime(11,00,00)));
firstEvent.setSensitivity(Sensitivity::Normal);
firstEvent.setAccountId(defaultCalendar.first);
firstEvent.setFolderId(defaultCalendar.second);
firstEvent.setSubject("Dentist");

// create first event in database
calendarService.createEvent(firstEvent);

CalendarEvent secondEvent;

// Create the attendees for the second event
Attendee firstAttendee;
Attendee secondAttendee;

firstAttendee.setName("John Smith");
firstAttendee.setRole(AttendeeRole::ReqParticipant);
```

```
secondAttendee.setName("Anwar Ludin");
secondAttendee.setRole(AttendeeRole::OptParticipant);

// Add the attendees to the second event, and specify other
// information for the event
secondEvent.setStartTime(QDateTime(QDate(2014, 03, 11), QTime(15, 0, 0)));
secondEvent.setEndTime(QDateTime(QDate(2014, 03, 11), QTime(18, 00, 0)));
secondEvent.setSensitivity(Sensitivity::Confidential);
secondEvent.setAccountId(defaultCalendar.first);
secondEvent.setFolderId(defaultCalendar.second);
secondEvent.setSubject("Annual Results");
QList<Attendee> attendees;
attendees << firstAttendee << secondAttendee;
secondEvent.setAttendees(attendees);

// Add the events to the database
calendarService.createEvent(secondEvent);
```

In practice, you should let the user choose the specific calendar where he wants to add the new event (for example, you could display a list of available calendars to the user by using the list returned by the `CalendarService:folders()` method; note that the method will also return remote calendars, which can be quite handy).

You can also use the `CalendarService::folders()` method to iterate by name over all of the user's calendars; for example, Listing 8-16 shows you how to add a new event in the user's "Hobbies" calendar.

Listing 8-16. Creating Events

```
#include <bb/pim/calendar/CalendarService>
#include <bb/pim/calendar/CalendarEvent>
#include <bb/pim/calendar/CalendarFolder>

using namespace bb::pim::calendar;

QList<CalendarFolder> folders = calendarService.folders();
foreach(CalendarFolder folder, folders){
    if(folder.name() == "Hobbies"){
        CalendarEvent sailingEvent;
        sailingEvent.setStartTime(QDateTime(QDate(2014, 03,12), QTime(14,00,00)));
        sailingEvent.setEndTime(QDateTime(QDate(2014, 03,12), QTime(18,30,00)));
        sailingEvent.setSensitivity(Sensitivity::Normal);
        sailingEvent.setAccountId(folder.accountId());
        sailingEvent.setFolderId(folder.id());
        sailingEvent.setSubject("Sailing competition");
        sailingEvent.setLocation("Geneva Yatch club");
        calendarService.createEvent(sailingEvent);
    }
}
```

Finally, you can check that the previous event has been successfully added to the Calendar app (see Figure 8-3). Besides, if the folder is linked to a `caldav` service provider, the corresponding event should also appear on the remote calendar.

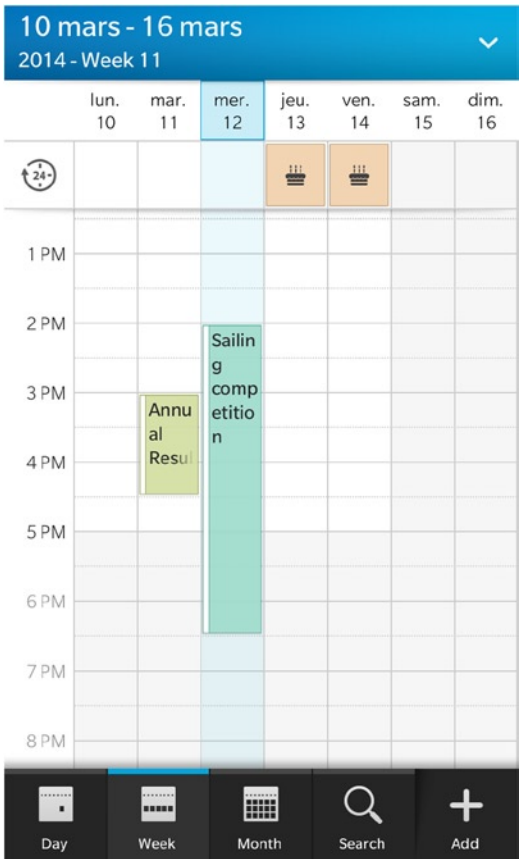

Figure 8-3. Events added to calendar

Searching for Calendar Events

You can define search criteria to search for particular events in a calendar by using the `EventSearchParameters` class.

The following summarizes important `EventSearchParameters` properties:

- `EventSearchParameters::setPrefix(const QString& prefix)`: Sets this search's prefix string. The search will return events whose subject or location string starts with the prefix string.
- `EventSearchParameters::setStart(const QDateTime& start)`: Sets the start date and time for this search.
- `EventSearchParameters::setEnd(const QDateTime& end)`: Sets the end date and time for this search.
- `EventSearchParameters::addFolder(const FolderKey& folder)`: Adds a folder key for this search. A `FolderKey` defines the account ID and folder ID to search (you can use `FolderKey::setAccountId(AccountId id)` and `FolderKey::setFolderId(FolderId id)` to define the calendar to be searched).

For example, Listing 8-17 shows you how to search the calendar database for the event created in Listing 8-16.

Listing 8-17. Searching Events

```
#include <bb/pim/calendar/CalendarService>
#include <bb/pim/calendar/CalendarEvent>
#include <bb/pim/calendar/EventSearchParameters>

using namespace bb::pim::calendar;

EventSearchParameters searchParams;
searchParams.setPrefix("sailing");
QList<CalendarEvent> events = calendarService.events(searchParams);
foreach(CalendarEvent event, events){
    qDebug() << "subjet: " << event.subject();
    qDebug() << "start time: " << event.startTime();
    qDebug() << "end time: "  << event.endTime();
}
```

Note that the prefix is case-insensitive and that the search will equally match “sailing” or “Sailing”.

Message API

The Message API enables you to send messages directly from your application. A message can include information such as subject, body, sender, and recipients. You can also include attachments to messages. Messages can take various forms, such as text or e-mail, and they can be grouped together in a conversation. Finally, some message types can be organized in folders (for example, Inbox, Sent, Trash, Deleted, and so on). A very convenient aspect of the Message API is that you can use a common interface to manage any kind of message, whether it is a text message or an e-mail. The Message API’s entry point is the `MessageService` class, which is described in the next section.

MessageService

`MessageService` is the interface to the messaging service. You can use `MessageService` to perform operations such as sending, saving, updating, removing, and retrieving messages. The following describes `MessageService` methods of interest:

- `MessageKey MessageService::send(bb::pim::account::AccountKey accountId, const Message& message)`: Sends a message. The `accountId` is given by `Account::id()`.
- `QList<Message> messages(bb::pim::account::AccountKey accountId, const MessageFilter& filter)`: Retrieves a list of messages using the search criteria given by filter.
- `int MessageService::messageCount(bb::pim::account::AccountKey accountId, const MessageFilter& filter)`: Returns the number of messages with the provided `accountId` and corresponding to the search criteria given by filter. You can use this method to predetermine the number of messages that will be returned using the search filter.

- `QList<MessageFolder> MessageService::folders(bb::pim::account::AccountKey accountId)`: Returns all message folders associated with this `accountId`.
- `bool MessageService::isFeatureSupported(bb::pim::account::AccountKey accountId, MessageServiceFeature::Type feature)`: Returns whether or not the indicated feature is supported by an account. In particular, you can use this method to determine if folder management is supported by passing `MessageServiceType::FolderManagement` to the method.
- `QList<Conversation> MessageService::conversation(bb::pim::account::AccountKey accountId, const MessageFilter& filter)`: Retrieves a list of conversations that fit the provided criteria.

Sending Messages

Sending messages, whether it is an e-mail or a short text message (SMS), is amazingly simple using the Message API. Listing 8-18 shows you the basic steps for creating a new message and sending it using the `MessageService` class.

Listing 8-18. Sending Messages

```
#include <bb/pim/account/AccountService>
#include <bb/pim/account/Account>

#include <bb/pim/message/Message>
#include <bb/pim/message/MessageBuilder>

AccountService accountService;
MessageService messageService;
QList<Account> accounts = accountService.accounts(Service::Messages, "imapemail");
if(accounts.size() > 0){
    Account account = accounts.first();  // use the first imapemail account available.

    MessageBuilder* builder = MessageBuilder::create(account.id());
    MessageContact recipient(-1, MessageContact::To, "Anwar Ludin", "anwar@aludin.com");

    builder->subject("Hello world");
    builder->body(MessageBody::PlainText, QString("This is the message body").toUtf8());
    builder->addRecipient(recipient);

    messageService.send(account.id(), *builder);

    delete builder;
}
```

Listing 8-18 creates a new `MessageBuilder` instance by passing an account corresponding to an `imapemail` service provider (as mentioned previously, you can potentially have multiple accounts corresponding to the same service provider and the code simply uses the first one returned by the `AccountService`). Next, you need to create a message recipient, which is represented by the `MessageContact` class and has to be added to the `MessageBuilder` instance. As illustrated, the

`MessageContact` instance is created using recipient's name, e-mail address, and the fact that he is the primary recipient (this is reflected by the `Message::To` parameter; if the message was copied, you should have used `Message::CC` instead). Finally, when all message parameters have been specified using the `MessageBuilder` instance, you can send the message using the `MessageService` instance.

Sending a short text message is similar to sending e-mails, except that you must use the `sms-mms` service provider and include your text message in a *conversation* (a conversation is essentially a grouping of related messages between recipients). The updated version of the code for sending text messages is shown in Listing 8-19.

Listing 8-19. Sending a Short Text Message

```
AccountService accountService;
MessageService messageService;

QList<Account> accounts = accountService.accounts(Service::Messages, "sms-mms");

    if(accounts.size() > 0){
        Account account = accounts.first();

        ConversationBuilder* conversationBuilder = ConversationBuilder::create();
        conversationBuilder->accountId(account.id());

        MessageContact recipient(-1, MessageContact::To, "Anwar Ludin", "0041766271***");

        QList<MessageContact> participants;
        participants << recipient;

        conversationBuilder->participants(participants);

        Conversation conversation = *conversationBuilder;
        ConversationKey conversationKey = messageService.save(account.id(), conversation);

        MessageBuilder* builder = MessageBuilder::create(account.id());

        builder->conversationId(conversationKey);

        builder->subject("Hello world");
        builder->body(MessageBody::PlainText, QString("This is the message body").toUtf8());
        builder->addRecipient(recipient);

        messageService.send(account.id(), *builder);

        delete conversationBuilder;
        delete builder;

    }
```

You can use the following `MessageService` signals to track new messages and message updates:

- `MessageService::messageAdded(bb::pim::account::AccountKey accountId, bb::pim::message::ConversationKey conversationId, bb::pim::message::MessageKey message)`: Emitted when a single message is added to the message service.
- `MessageService::messageUpdated(bb::pim::account::AccountKey accountId, bb::pim::message::ConversationKey conversationId, bb::pim::message::MessageKey messageId, bb::pim::message::MessageUpdateData data)`: Emitted when a message is updated in the message service.

Searching for Messages

You can use the message service to search for messages corresponding to specific search criteria. For example, you can specify that you are interested in messages sent to a specific recipient or messages containing a given text in their body. You can also search messages by status. To perform a search, you need to use a `MessageSearchFilter` instance:

- `MessageSearchFilter::addSearchCritera(SearchFilterCriteria::Type criteria, const QString& value)`: Adds a search criteria to this message search filter.
- `MessageSearchFilter::addStatusCriteria(SearchStatusCriteria::Type criteria)`: Adds a status criteria to this message search filter. For example, if you want to apply the search to inbound (received) messages, you can use `SearchStatusCriteria::Received`.

Listing 8-20 illustrates how to use a search filter in practice.

Listing 8-20. Searching Messages

```
// Create the message service object
MessageService service;

// Create the search criteria
MessageSearchFilter filter;
filter.addSearchCriteria(SearchFilterCriteria::Subject, "BlackBerry 10 book");
filter.addSearchCriteria(SearchFilterCriteria::Body, "Chapter 8");
filter.addStatusCriteria(SearchStatusCriteria::Received);
filter.setLimit(20);

// Perform a local search using the filter criteria
QList<Message> localMessageResults = service.searchLocal(1, filter);

// Perform a remote search using the filter criteria
QList<Message> remoteMessageResults = service.searchRemote(1, filter);
```

As illustrated in Listing 8-20, you can also specify whether the search should be performed locally on the device or remotely on the messaging server.

Message API Summary

This section provides you with a brief summary of the Message APIs.

MessageBuilder

The `MessageBuilder` class lets you create a new `Message` object or edit an existing one. The following summarizes important `MessageBuilder` methods:

- `MessageBuilder& MessageBuilder::addRecipient(const MessageContact& recipient, bool* ok=0)`: Adds the recipient to the message. You can check if the operation was successful by using the `ok` flag.
- `MessageBuilder& MessageBuilder::body(MessageBody::Type, const QByteArray& data)`: Sets the body of this message, which can be either plain text (`MessageBody::PlainText`) or HTML (`MessageBody::Html`).
- `MessageBuilder& MessageBuilder::addAttachment(const Attachment& attachment, bool* ok=0)`: Adds an attachment to this message.
- `MessageBuilder::operator Message()`: Casts this `MessageBuilder` into a `Message`.

MessageContact

A `MessageContact` object represents a recipient or sender of a message and includes the contact id, contact type, name, and e-mail address. The following summarizes `MessageContact` methods of interest:

- `MessageContact::MessageContact(MessageContactKey, MessageContact::Type type, const QString& name, const QString& address, unsigned char ton=0, unsigned char npi=0)`: Constructs a message contact. `MessageContactKey` corresponds to the id of a Contact retrieved from the Contacts database. You can set this value to –1 if the message contact is not located in the Contacts database. `MessageContact::Type` can take the following values: `MessageContact::To`, `MessageContact::Cc`, `MessageContact::Bcc`, `MessageContact::From`, and `MessageContact::ReplyTo`. The last two parameters are optional and are used only in alphanumeric addresses in SMS. Finally, in the case of SMS messages, you can simply pass the contact phone number in the name and address fields.
- `QString MessageContact::displayableName()`: Returns the displayable name of this contact, which includes the contact name, friendly name, and e-mail address.

ConversationBuilder

A conversation is a set of related messages between recipients. The main purpose of organizing messages in conversations is to display them together in your UI (for example, as a thread of related messages). The following summarizes important `ConversationBuilder` methods:

- `ConversationBuilder* ConversationBuilder::create()`: Starts a new conversation.
- `ConversationBuilder& ConversationBuilder::accountId(bb::pim::account::AccountKey accountId)`: Associates this conversation with the user account given by `accountId`.
- `ConversationBuilder& ConversationBuilder::name(QString string)`: Sets the name of this conversation.
- `ConversationBuilder& ConversationBuilder::participants(QList<MessageContact> participants)`: Sets the participants of this conversation.
- `ConversationBuilder::operator Conversation()`: Casts this `ConversationBuilder` into a `Conversation` object.

Summary

Personal information management (PIM) is an important aspect of writing applications for the BlackBerry 10 platform. This chapter reviewed the BlackBerry 10 PIM APIs and showed you how to use them in your own applications. The APIs provide a service interface, which can be used to update and search the corresponding PIM data stores. A PIM data store contains items such as contacts, calendars, messages, and notebooks. The BlackBerry 10 PIM APIs use service types such as `Messages`, `Calendars`, and `Contacts` to describe groups of services. Service providers provide the actual implementation. The service providers are in turn linked to accounts on the device, which provide access to the target systems. This chapter covered mostly the PIM service providers, but in practice, the BlackBerry 10 device uses a wide range of service providers (such as social networking providers, for example).