Chapter 7

HTTP Networking

HTTP networking is ubiquitous on mobile devices. This book would certainly not be complete if it did not include a chapter explaining how to use the BlackBerry 10 networking services. In this chapter, I am going to exclusively concentrate on HTTP networking, which covers about 90 percent of the cases you will face during application development. Also, BlackBerry 10 leverages the underlying QtNetwork module, which makes HTTP programming amazingly simple. The goal of this chapter is to show you how the different networking classes work together to access HTTP servers from a BlackBerry 10 mobile device.

An immediate application of networking is obviously to build a "rich thin client" where you use Cascades to build your application's native user interface and remotely access business logic implemented as rest services. By now you must have realized that Cascades and QML make user interface design a snap. Adding networking to the mix just opens a completely new dimension of connected applications. For example, exposing enterprise services securely to your workforce—something that BlackBerry has always been at the forefront with—is an obvious practical application.

After having read this chapter, you will have a good understanding of

- The Qt networking classes.
- How to use the networking classes to build connected Cascades applications.
- How to design responsive UIs by handling network requests and replies asynchronously.

Another important goal of this chapter is to illustrate all the concepts introduced so far by writing a slightly more complex app than the ones demonstrated so far. The application will take the form of a Cascades client app for a remote weather REST service and will emphasize the separation of UI logic from the core business logic written in C++. The application will also show you how to breakdown your C++ code in classes with delimited responsibilities.

Qt Networking Classes

HTTP networking using Qt mostly involves the following classes:

- QNetworkAccessManager: This class allows you to send network requests and receive replies. The QNetworkAccessManager's API is entirely asynchronous, thus guaranteeing that the user interface thread is not blocked during an HTTP request.
- QNetworkRequest: This class encapsulates all the required information for an HTTP request. Typically, you will be using QNetworkRequest's url property to access an HTTP URL.
- QNetworkReply: This is QNetworkRequest's counterpart; it encapsulates the data received from the server.

QNetworkAccessManager

QNetworkAccessManager is the grand dispatcher of all the network interactions in your application. You will generally use a single instance of this class to handle all the networking logic of your app. The QNetworkAccessManager object holds the common configuration and settings for the requests it sends. It should be noted that all functions in this class are *reentrant*. This means that you can call the class methods multiple times, even if a given network request has not yet completed (this is also possible because the class methods are asynchronous, or in other words, nonblocking). If necessary, the QtNetworkAccessManager internally queues the requests it receives, but has the capability to process multiple requests concurrently. The following is a review of QNetworkAccessManager's most important methods:

- QNetworkReply* QNetworkAccessManager::get(const QNetworkRequest& request): Posts a request to obtain the contents of the target specified by request. For the HTTP protocol, the request corresponds to the HTTP GET request. Returns a pointer to a QNetworkReply object, opened for reading, which can be used to retrieve data as soon as it is available.
- QNetworkReply* QNetworkAccessManager::post(const QNetworkRequest& request, const QByteArray& data): Sends an HTTP POST request to the destination specified by request and returns a pointer to a QNetworkReply object opened for reading. QNetworkReply contains the server's response. The QByteArray instance contains the data to be uploaded to the server.
- QNetworkReply* QNetworkAccessManager::post(const QNetworkRequest& request, QIODevice* data): Similar to the previous method, but this time the posted data is passed as a pointer to a QIODevice object. In other words, you can use this method to post the contents of a file by passing a QFile object as the second method parameter (this is possible because QFile inherits from QIODevice).
- QNetworkReply* QNetworkAccessManager::post(const QNetworkRequest& request, QHttpMultipart* multipart): Posts the content of a multipart message to the destination identified by request.

- `QNetworkReply* QNetworkAccessManager::put(const QNetworkRequest& request, const QByteArray& data)`: Sends an `HTTP PUT` request to the destination specified by `request` and returns a pointer to a `QNetworkReply` object opened for reading. This method makes sense in the context of a `REST` service, where `PUT` is used for creating a resource and `POST` for updating or modifying one. The `QNetworkReply` object contains the optional server response. The `QByteArray` instance contains the data to be uploaded to the server. Just as with an `HTTP POST` request, the method is overloaded and can also take a `QIODevice*` and `QHttpMultipart*` as a second parameter.
- `QNetworkConfiguration QNetworkAccessManager::configuration()`: Returns the network configuration that will be used to create the network session.
- `void QNetworkAccessManager::setConfiguration(const QNetworkConfiguration& config)`: Sets the network configuration that will be used to create the network session.
- `QNetworkCookieJar QNetworkAccessManager::cookieJar()`: Returns an instance of `QNetworkCookieJar` used to store cookies obtained from the network, as well as cookies about to be sent.
- `void QNetworkAccessManager::setCookieJar(QNetworkCookieJar* cookieJar)`: Sets the manager's cookie jar. The cookie jar will be used by all requests dispatched by the network manager.
- `void QNetworkAccessManager::setCache(QAbstractNetworkCache* cache)`: Sets the network manager's cache. The cache is used for all requests dispatched by the manager. You can use this function to specify an object that implements additional features, such as saving cookies to permanent storage or caching JavaScript and CSS files. Note that, by default, the network manager does not cache data. `QAbstractNetworkCache` provides the interface for cache implementation. As implied by its name, `QAbstractNetworkCache` is an abstract base class that cannot be instantiated. Instead, you can use a `QNetworkDiskCache`, which provides a concrete implementation. You can also control cache configuration with the `QNetworkRequest` request object (this will be explained in the next section).

Note As mentioned previously, you should always reuse the same QNetworkAccessManager instance. Note that you can conveniently access the default declarative engine's QNetworkAccessManager instance by using the `QMLDocument::defaultDeclarativeEngine()->networkAccessManager()` method call (because `QMLDocument::defaultDeclarativeEngine()` is a static method, you can always access the associated default declarative engine from anywhere in your code).

QNetworkRequest

A QNetworkRequest object holds a URL to be requested by a QNetworkAccessManager. You can specify the target URL using one of the following methods:

- QNetworkRequest::QNetworkRequest(const QUrl& url = QUrl()): Constructs a new network request with url as the URL to be requested.
- QNetworkRequest::setURL(const QUrl& url): Sets the URL this network request is referring to.

You can also provide additional information to further customize the request (for example, by setting header values, request priorities, and cache configurations). In the specific case of caching, you can specify the cache behavior by setting a QNetworkRequest's CacheLoadControlAttribute attribute, as follows:

- QNetworkRequest::setAttribute(QNetworkRequest::CacheLoadControlAttribute, const QVariant& value): Sets the cache behavior. The following are the possible values:
  - QNetworkRequest::AlwaysNetwork: Always load from the network and do not check if the cache has a valid entry.
  - QNetworkRequest::PreferNetwork: This is the default behavior; load from the network if the cache entry is older.
  - QNetworkRequest::PreferCache: Load from the cache first; otherwise, load from the network. Note that you risk loading stale data in this case.
  - QNetworkRequest::AlwaysCache: Always try to load from the cache. In other words, this option corresponds to an offline mode. Note that you can use QNetworkRequest::PreferCache for specific file types, such as CSS and JavaScript, where you are certain that they will not change during the application's lifetime.

Because you can specify the cache behavior on a *per request* basis, this can be very convenient if you have multiple requests of different kinds. However, for the biggest majority of network requests, you can simply set the target URL and pass the request to the QNetworkAccessManager.

QNetworkReply

QNetworkReply encapsulates the server's response and provides all the necessary functionality for retrieving the received data. The class inherits from QIODevice, which is the abstract base class for devices supporting reading and writing blocks of data. You will generally use the QByteArray QIODevice::read(qint64 maxSize) and QByteArray QIODevice::readAll() methods to retrieve the data. The former method reads, at most, maxSize bytes from the device. The latter reads all available data from the device. Both methods return the data as a QByteArray.

The following summarizes QNetworkReply's most important methods:

- bool QNetworkReply::isRunning() const: Returns true if the corresponding request is still being processed.
- QByteArray QNetworkReply::read(qint64 maxSize): Inherited from QIODevice; see description given at the start of this section.

- `QByteArray QNetworkReply::readAll()`: Inherited from `QIODevice`; see description given at the start of this section.
- `QNetworkRequest QNetworkReply::request()`: Returns the request that was posted for this reply.
- `QUrl QNetworkReply::url()`: Returns the `URL` of the content downloaded or uploaded. Note that the `URL` may be different from the one specified in the original request.
- `NetworkError QNetworkReply::error()`: Returns the error that was found during the processing of this request. Returns `QNetworkReply::NoError` if the request was processed successfully. Check the API documentation for all the possible values taken by the `QNetworkReply::NetworkError` enumeration.
- `QVariant QNetworkReply::attribute(Attribute code, const QVariant& defaultValue = QVariant())`: Returns the attribute associated with `code`. If `code` has not been set, returns `defaultValue`. Attributes are metadata that are used to pass additional information from the reply back to the application. As you will see in the examples section, you will use this property to detect `HTTP` redirects.
- `QNetworkReply::abort()`: Aborts the operation immediately and closes any network connections still open.

`QNetworkReply` can also emit the following signals:

- `QNetworkReply::finished()`: This signal is emitted when the reply has finished processing. The data can be retrieved by calls to `QNetworkReply::read()` or `QNetworkReply::readAll()`.
- `QNetworkReply::downloadProgress(qint64 bytesReceived, qint64 bytesTotal)`: This signal is emitted to indicate the data download's progress for a given network request. The download is finished when `bytesReceived` is equal to `bytesTotal`. Note that you should handle this signal when large amounts of data are being downloaded to convey some feedback to the user (for example, by displaying a Cascades `ProgressIndicator`). (You can also opt to process the data in chunks, as it becomes available.) The `bytesReceived` parameter indicates the number of bytes received, whereas `bytesTotal` indicates the total number of bytes expected to be downloaded. Note that if the total number of bytes to be downloaded is unknown, `bytesTotal` will be –1, but when the download has completed `bytesReceived` will always be equal to `bytesTotal`.
- `QNetworkReply::uploadProgress(qint64 bytesSent, qint64 bytesTotal)`: This signal is emitted to indicate the upload progress of a network request. The upload is finished when `bytesSent` is equal to `bytesTotal`.
- `QNetworkReply::sslErrors(const QList<QSslError>& errors)`: This signal is emitted if the SSL/TLS session encountered errors during the setup, including certificate verification errors. The list of errors is provided by the errors parameter.

> **Note** You should always warn the user if ssl errors occur and give him the option to cancel the request.

HTTP Networking Examples

The examples provided in this section illustrate typical usage scenarios of the networking classes.

HTTP GET

Let's start with a simple GET request to access a REST service. The data in the response will be returned in JSON format. To parse the object, you will have to use an instance of the Cascades JsonDataAccess class and handle the JSON structure in-memory. The Qt object constructed from JSON by the JsonDataAccess instance will always be a QVariant that either contains a QVariantList (if an array of JSON objects is returned by the service) or a QVariantMap (if a single object is returned). The mapping between JSON types and Qt types is summarized as follows:

- int: Mapped to a QVariant(Int64). To access the contained int use QVariant::toInt().
- uint: Mapped to a QVariant(Uint64). To access the contained uint use QVariant::toUInt().
- real: Mapped to a QVariant(double). To access the contained real use QVariant::toReal().
- string: Mapped to a QVariant(const char*). To access the contained string use QVariant::toString().
- boolean: Mapped to a QVariant(bool). To access the contained boolean use QVariant::toBool().
- array: Mapped to a QVariant(QVariantList). To access the contained array use Qvariant::toList().
- object: mapped to a QVariant(QVariantMap). To access the contained object, use QVariant::toMap().

The requested URL corresponds to the list of categories defined in my WordPress blog and is given at http://aludin.com?json=get_category_index. Listing 7-1 shows you an example of the returned JSON object.

Listing 7-1. JSON Response

```
{
    "status": "ok",
    "count": 2,
    "categories": [
        {
            "id": 2,
            "slug": "lifeinit",
```

```
            "title": "Life in IT, Anti-Patterns of Efficiency",
            "description": "",
            "parent": 0,
            "post_count": 2
        },
        {
            "id": 3,
            "slug": "mobile-computing",
            "title": "Mobile Computing",
            "description": "",
            "parent": 0,
            "post_count": 1
        }
    ]
}
```

Listing 7-2 shows you how to perform the `HTTP GET` request to retrieve the JSON document displayed in Listing 7-1.

Listing 7-2. ApplicationUI::getCategories()

```
ApplicationUI::getCategories(){
    QString url("http://aludin.com?json=get_category_index");
    QNetworkRequest request(url);

    QNetworkReply* reply = this->m_networkManager->get(request);
    bool result = connect(reply, SIGNAL(finished()), this,
                          SLOT(onCategoriesFinished()));
    Q_ASSERT(result);
}
```

It is not shown in the previous code, but you can safely assume that `ApplicationUI::my_networkManager` has been initialized with the default declarative engine's `QNetworkAccessManager`.

And Listing 7-3 illustrates how to perform the actual `JSON` response parsing once it has been returned by the service.

Listing 7-3. ApplicationUI::onCategoriesFinished()

```
void ApplicationUI::onCategoriesFinished() {
    QNetworkReply* reply = static_cast<QNetworkReply*>(QObject::sender());
    if (!reply->error()) {
        JsonDataAccess jda;
        QVariant response = jda.load(reply);
        QVariantMap map = response.toMap(); // get root JSON object
        QString statusValue = map["status"].toString();
        QVariantList categories = map["categories"].toList(); // get categories array.
        for(int i=0; i<categories.size(); i++){
           QString title = categories[i].toMap()["title"].toString();
        }
    }
    reply->deleteLater();
}
```

You will see later that you can conveniently chain the QVariant method calls to navigate the JSON object structure. Note that as a convenience and for clarity, I am using strings literals directly in the code, but ideally you should use string constants to avoid sprinkling your code with literals.

Finally, if your request takes additional parameters, you should use URL encoding to make sure that the parameters do not contain reserved HTTP characters (see Listing 7-4).

Listing 7-4. URL Percent-Encoding

```
QString date("50-2010/05/11 22:45:19 +0000");
QString encodedDate = QString(QUrl::toPercentEncoding(date));
QString getUrl = QString("http://www.aservice.com");
getUrl.append("?date=");
getUrl.append(encodedDate);
```

HTTP POST

Posting data is just as simple as performing HTTP GET requests. You will have to specify the data parameters by adding them to a QByteArray. You also need to make sure that you separate each parameter-value pair with an ampersand, as shown in Listing 7-5.

Listing 7-5. Post Example

```
void ApplicationUI::doPost(){
     // Setup the webservice url
     QUrl postUrl = QUrl("http://www.aservice.com");
     QByteArray postData;

     postData.append("param1=value1&").append("param2=value2&").append("param3=value3");

    // Call the webservice
    QNetworkReply* reply = this->m_networkManager->post(QNetworkRequest(postUrl), postData);
    bool result = connect(reply, SIGNAL(finished()), this,
                          SLOT(onPostFinished()));
    Q_ASSERT(result);
}
```

Once again, in practice you should use percent-encoding for the parameters you pass to the POST request. Also, in the onPostFinished() slot, don't forget to release the QNetworkReply instance using QNetworkReply::deleteLater().

Handling an HTTP Redirect

At certain times, you will have to process an HTTP redirect. A redirect is not an error and simply indicates that a resource has moved. Listing 7-6 shows you how to handle the situation.

Listing 7-6. Redirect Check Example

```
void ApplicationUI::onRequestFinished(QNetworkReply* reply){
    if(reply->error() == QNetworkReply::NoError){
        QVariant redirect =
        reply->attribute(QNetworkRequest::RedirectionTargetAttribute);
        if(!redirect.isNull()){
            QUrl originalUrl = reply->request().url();
            QUrl newUrl = originalUrl.resolved(redirect.toUrl());
            // send new network request using newUrl
        }else{
            // process data
        }
    }else{
        // handle error in response
    }
    reply->deleteLater();
}
```

In practice, you should always be ready to handle HTTP redirects.

Handling Authentication

Certain HTTP services will require authentication before providing you access to their resources. In those cases, you can use the `QNetworkAccessManager::authenticationRequired(QNetworkReply* reply, QAuthenticator* authenticator)` signal to handle the authentication request. Listings 7-7 and 7-8 illustrate how to implement authentication in your own code.

Listing 7-7. ApplicationUI.hpp

```
ApplicationUI::ApplicationUI(bb::cascades::Application *app) :
    QObject(app),
    m_networkManager(QMLDocument::defaultDeclarativeEngine->networkAccessManager())
{
    bool result = connect(m_networkManager,
                    SIGNAL(authenticationRequired(QNetworkReply*, QAuthenticator*)), this,
                    SLOT(onAuthenticationRequired(QNetworkReply*, QAuthenticator*)));
    Q_ASSERT(result);
}
```

Listing 7-8. ApplicationUI.cpp Authentication Handler

```
void ApplicationUI::onAuthenticationRequired(QNetworkReply* reply,
                                             QAuthenticator* authenticator)
{
    SystemCredentialsPrompt prompt = new SystemCredentialsPrompt;
    prompt->exec();
    authenticator->setUser(prompt->usernameEntry());
    authenticator->setPassword(prompt->passwordEntry());
    prompt->deleteLater();
}
```

The `QNetworkAccessManager::authenticationRequired(QNetworkReply*, QAuthenticator*)` signal is connected to the corresponding slot in the application delegate's constructor. Therefore, whenever a server request needs to be authenticated, the slot will be called. As shown in Listing 7-8, you can use a `SystemCredentialsPrompt` object to display a modal dialog requesting the user's credentials (see Figure 7-1). Note that the majority of Cascades controls methods are nonblocking (in other words, they return immediately and processing continues). However, in this specific case, we want to be able to call a blocking method until the user has provided his credentials. To achieve this behavior, you should call `SystemCredentialsPrompt::exec()` instead of `SystemCredentialPrompt::show()`, which is the nonblocking version. (Internally, `SystemCredentialsPrompt::exec()` creates a nested event loop to provide the blocking functionality. When the nested event loop is exited, control is returned to the main event loop). Note that once you have finished with the prompt object, you must call `QObject::deleteLater()` instead of deleting the object immediately.

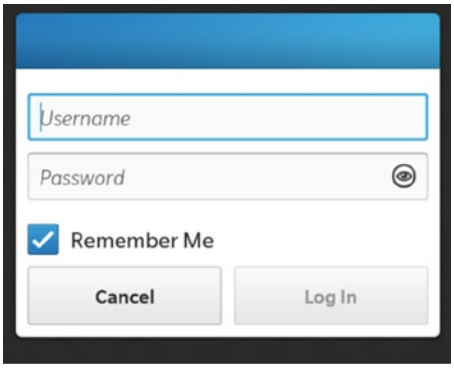

Figure 7-1. *Credentials prompt*

Finally, the authenticator should be updated with the user's credentials, which are sent back to the server.

Weather2

I promised you in Chapter 2 that we would build a weather app relying on the `REST` service introduced at the time. In essence, I want to illustrate how you can design an enticing Cascades UI on top of raw data (which would be the `JSON` document returned by the weather service). You will also learn how to combine multiple services together (such as Google Maps) to further enrich your application. Finally, you will see how the networking classes are used in practice to perform asynchronous requests. The application we are about to design is called, quite appropriately, Weather2 (the default Weather app is bundled with BlackBerry 10). The finished application's UI is shown in Figures 7-2, 7-3, and 7-4. The application has two tabs. On the first tab, you can perform a query by country, state, or city using a text field. If your query returns multiple results, the application will ask you to select a city from a list of values displayed in a `SystemListDialog` (see Figure 7-2).

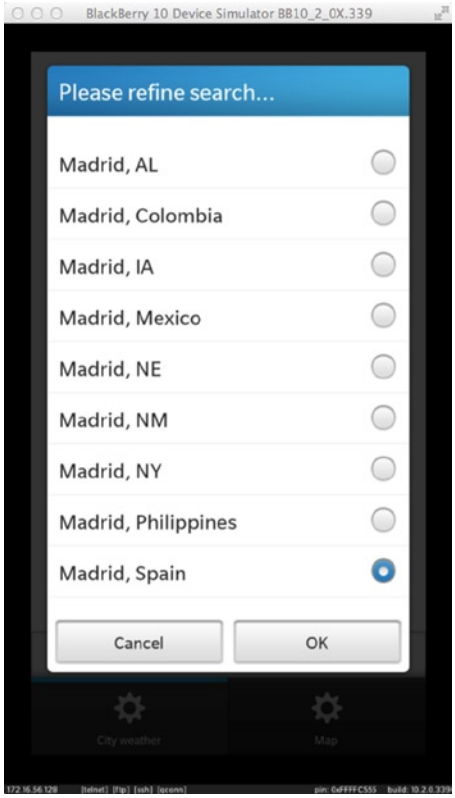

***Figure 7-2.** City selection*

As soon as you have selected a city, the weather conditions are displayed, including the city's latitude and longitude (see Figure 7-3).

Figure 7-3. *City view*

If you select the second tab, a map will be displayed, with the city location highlighted by a small icon representing the weather conditions (see Figure 7-4).

Figure 7-4. Map view

Application Design

Before actually looking at the app implementation, let's summarize once again the most important BlackBerry 10 design principles and recommendations (you can refer to Chapter 3 for a more detailed discussion of these points):

- Separate UI logic from business logic. Although it is possible to directly access Cascades controls from C++, the preferable way to build BlackBerry 10 apps is by clearly decoupling the UI logic from the rest of the application's logic written in C++. As stated in Chapter 3, one of the major strengths of QML and C++ integration is the ability to implement the QML UI separately from C++. The C++ business logic can therefore be blissfully unaware of the QML layer (in other words, using `QObject::findChildren()`to access Cascades controls by object name from C++ is considered a bad practice because it adds tight coupling between UI and business logic).

- Prefer signals for communicating between QML and C++.
- Prefer properties and QML bindings to synch data between QML and C++ (you will also notice that at times I pass data as signal parameters). A QML component can have its properties bound to a C++ class' properties. If a C++ property is updated, a signal has to be emitted from C++ in order notify the QML declarative engine, which then updates the corresponding QML bound property. Note that bindings can be defined both ways: the declarative engine will also automatically update the C++ bound property when the corresponding QML property changes.
- Break down your UI in multiple QML components instead of designing it as a single monolithic bloc. This will save you major headaches when you need to selectively update UI parts. Indeed, the ability to extend QML with your own custom components is a major advantage that you should leverage as much as possible.

Having emphasized these points, let's start with the UI design.

> **Note** The source code for the Weather2 application can be found in this book's repository on GitHub at https://github.com/aludin/BB10Apress.

Creating the UI

Weather2's UI is split between four QML components:

- `main.qml`: The QML document initially loaded by the application delegate. It defines a tabbed pane containing two tabs (see Listing 7-9).
- `WeatherDetails.qml`: The control responsible for handling user input for weather requests. The control also manages various system prompts for notifying or requesting additional information from the user, when necessary (you will see that the prompts are defined as attached objects).
- `City.qml`: The control responsible for displaying the weather data for a given city. Note that this control is referenced in `WeatherDetails.qml` (see Listing 7-10).
- `WeatherMap.qml`: The control responsible for displaying a map with the weather conditions for the selected city (see Listing 7-11).

Listing 7-9. main.qml

```
import bb.cascades 1.2
TabbedPane {
    id: tabbedPane
    showTabsOnActionBar: true
```

```
    Tab {
        title: "City weather"
        Page {
            WeatherDetails {
                // control loaded from WeatherDetails.qml
            }
        }
    }
    Tab {
        title: "Map"
        Page {
            WeatherMap {
                // control loaded from WeatherMap.qml
            }
        }
    }
}
```

As you can see in Listing 7-9, the `WeatherDetails` and `WeatherMap` controls are used as content properties for page controls. The QML engine will therefore automatically load the controls from the corresponding files located in the assets folder of your application project (note that `WeatherDetails.qml` and `WeatherMap.qml` are located in the same folder as `main.qml`).

Let us now have a look at the WeatherDetails control implementation (see Listing 7-10).

Listing 7-10. WeatherDetails.qml

```
import bb.cascades 1.2
import bb.system 1.2
Container {
    id: main
    background: back.imagePaint
    function onError(message) {
        errorPrompt.title = message;
        errorPrompt.show();
    }

    function onMultipleCitiesFound(cities) {
        citiesDialog.clearList();
        for (var i = 0; i < cities.length; i ++) {
            citiesDialog.appendItem(cities[i]);
        }
        citiesDialog.show();
    }

    function onFinished() {
        progress.cancel();
    }
```

```
    onCreationCompleted: {
        _app.weather.multipleCitiesFound.connect(main.onMultipleCitiesFound);
        _app.weather.error.connect(main.onError);
        _app.weather.finished.connect(main.onFinished);
        progress.cancelButton.label = "Cancel";
        progress.confirmButton.label = "";
    }

    attachedObjects: [
        ImagePaintDefinition {
            id: back
            repeatPattern: RepeatPattern.XY
            imageSource: "asset:///images/background.jpg"
        },
        SystemListDialog {
            id: citiesDialog
            onFinished: {
                if (value == SystemUiResult.ConfirmButtonSelection) {
                    _app.weather.cityWeather(citiesDialog.selectedIndices[0]);
                    progress.show();
                }
            }
        },
        SystemPrompt {
            id: errorPrompt
            onFinished: {
                _app.weather.cityWeather(errorPrompt.inputFieldTextEntry());
                progress.show();
            }
        },
        SystemProgressDialog {
            id: progress
            title: "Retrieving city"
            onFinished: {
                if (value == SystemUiResult.CancelButtonSelection) {
                    _app.weather.cancel();
                }
            }
        }
    ]
    layout: StackLayout {
        orientation: LayoutOrientation.BottomToTop
    }
    TextField {
        id: location
        inputMode: TextFieldInputMode.Default
        textStyle.textAlign: TextAlign.Center
        input {
            submitKey: SubmitKey.Go
            submitKeyFocusBehavior: SubmitKeyFocusBehavior.Lose
```

```
            onSubmitted: {
                _app.weather.cityWeather(location.text);
                progress.show();
            }
        }
        hintText: "Enter city or country name"
    }
    City{
        // control loaded from City.qml
    }
}
```

As you can see, `WeatherDetails.qml` mostly contains some JavaScript code responsible for signal handling. Also, an important point to consider is the way the emitted signals from C++ are connected to the JavaScript functions in the main container's `onCreationCompleted` slot (in other words, the `onError()`, `onMultipleCitiesFound()`, and `onFinished()` JavaScript functions or slots for signals emitted by the `_app.weather` C++ object). Also note how the location text field's `onSubmitted` slot is used for calling the `_app.weather.cityWeather()` slot, which is defined in C++. If the user's initial query returns multiple cities, a `SystemListDialog` is displayed, asking him to further refine the query. In the same manner, if an error occurs because the user's query is incorrect, a `SystemPrompt` is displayed, asking him to correct the query. In both cases, `_app.weather.cityWeather()` is called with the user's updated query.

The `City` control is mostly a visual control for displaying the results of a weather request: the control uses labels and an image view for displaying the weather conditions for a given city. All QML properties defined in the control are bound to corresponding C++ properties (for example, Listing 7-11 gives you the binding for the current temperature).

Listing 7-11. City Control, Binding Example

```
Label {
    id: temperature
    text: _app.weather.cityinfo.temperature
    horizontalAlignment: HorizontalAlignment.Center
    textStyle {
        fontWeight: FontWeight.W100
        color: Color.Black
        fontSize: FontSize.PercentageValue
        fontSizeValue: 250
    }
}
```

In the example provided in Listing 7-11, the label's `text` property is bound to the `_app.weather.cityinfo.temperature` property, which is defined in C++ (as you will see in a moment). Therefore, when the `_app.weather.cityinfo.temperature` property is updated in C++, the QML declarative engine automatically updates the label's text property.

The final QML component to consider is the `WeatherMap` component, which appears on the second tab. Listing 7-12 gives you component definition.

Listing 7-12. WeatherMap Control

```
import bb.cascades 1.2
import ludin.utils 1.0
Container {
    layout: DockLayout {
    }
    onCreationCompleted: {
        _app.weather.cityinfo.coordinatesChanged.connect(mapclient.setCoordinates);
        scrollview.zoomToPoint(320, 220, 2, ScrollAnimation.Smooth);
    }
    attachedObjects: [
        GoogleMapClient {
            id: mapclient
        }
    ]
    ScrollView {
        id: scrollview
        horizontalAlignment: HorizontalAlignment.Fill
        verticalAlignment: VerticalAlignment.Fill
        scrollViewProperties {
            scrollMode: ScrollMode.Both
            pinchToZoomEnabled: true
        }
        ImageView {
            id: citymap
            image: mapclient.image
        }
    }
}
```

Here again, the control is relatively simple. It mainly consists of an image view responsible for displaying a map of the current weather conditions for a given location. The `GoogleMapClient` attached object provides the actual weather image. Once again, QML bindings are used to synch the image view and the image map generated by the `GoogleMapClient` attached object. Finally, the current map coordinates are provided to the `GoogleMapClient` attached object by the `_app.weather.cityinfo.coordinatesChanged()` signal (the signal to the slot connection is done in the main container's `onCreationCompleted` slot).

Adding the C++ Implementation

Let us now turn our attention to the C++ implementation. The most important factor to consider is how to organize your code so that you can define classes with specific responsibilities:

- `WeatherClient`: Responsible for performing the REST requests to the Weather Underground service (www.wunderground.com/weather/api). The class also handles the parsing of the JSON response.
- `CityInfo`: Encapsulates the weather data once it has been returned by the Weather Underground service. Note that the QML `City` control has its properties bound to `CityInfo`'s properties.

- `GoogleMapClient`: A client for generating static maps using the Google Maps service. An instance of this class is defined as an attached object property of the `WeatherMap` control.
- `ApplicationUI`: The standard application delegate reachable from the QML layer of your application through the QML document context.

The class relationships are also quite simple: the `ApplicationUI` object has a `WeatherClient weather` property, which in turn has a `CityInfo` property. The properties are accessible from QML as `_app.weather` and `_app.weather.cityinfo`, respectively.

WeatherClient

The `WeatherClient` class definition is given in Listing 7-13.

Listing 7-13. WeatherClient Class Definition

```
#ifndef WEATHERCLIENT_H_
#define WEATHERCLIENT_H_

#include <QObject>

#include <QNetworkAccessManager>
#include <QNetworkReply>

#include  "CityInfo.h"
#include "GoogleMapClient.h"

class WeatherClient : public QObject {
    Q_OBJECT
    Q_PROPERTY(CityInfo* cityinfo READ city CONSTANT)
public:
    WeatherClient(QObject* parent=0);
    virtual ~WeatherClient();

signals:
    void multipleCitiesFound(QStringList cities);
    void keyError(const QString& message
    void error(const QString& message);
    void finished();

public slots:
    void cityWeather(QString city);
    void cityWeather(int selectedIndex);
    void cancel();

private slots:
    void onCityRequestFinished();
    void onCategoriesFinished();
private:
    CityInfo* city() const;
```

```
    void updateCityInfo(const QVariantMap& map);

    QString m_apiKey;
    QNetworkAccessManager* m_networkManager;
    QList<QNetworkReply*> m_networkReplies;
    CityInfo* m_cityInfo;
    QStringList m_cities;

   // static char* constant tags omitted
};

#endif /* WEATHERCLIENT_H_ */
```

The class definition declares multiple slots and signals. To perform an initial weather request, the `WeatherClient::cityWeather(QString city)` slot has to be called from QML (you might recall from Chapter 3 that C++ slots and functions marked as `Q_INVOKABLE` can be called from QML). Also note that the signals are the same as those handled in JavaScript by the `WeatherDetails` control (see Listing 7-10). The `multipleCitiesFound` signal is emitted when a user query corresponds to multiple cities. (The cities are stored in a `QStringList` and passed as a parameter to the signal. As soon as the user selects a specific city, the `WeatherClient::cityWeather(int selectedIndex)` slot is called from QML and a new request is sent to the weather service.) The `error` signal is emitted when the Weather Underground service returns an error (the error is passed as a `QString` parameter to the signal), and, finally, the `finished` signal is emitted when a network request has completed.

Let us now turn our attention to the `WeatherClient` member function definitions.

Constructor

Listing 7-14 gives you the WeatherClient constructor.

Listing 7-14. WeatherClient Constructor

```
WeatherClient::WeatherClient(QObject* parent) :
    QObject(parent),
    m_networkManager(QmlDocument::defaultDeclarativeEngine()->networkAccessManager()),
    m_cityInfo(new CityInfo(this))
{
    JsonDataAccess jda;
    QVariant keyMap = jda.load(
    QDir::currentPath() + WeatherClient::m_apiKeyPath);

    if (jda.hasError()) {
        emit keyError("Error, could not read api key");
    } else {
        m_apiKey = keyMap.toMap()[WeatherClient::m_keyTag].toString();
    }
}
```

The constructor proceeds by initializing the class members using a member initialization list. The constructor body then tries to load the Weather Underground API key, which is required for each service request. The API key is stored in a JSON file located in a subfolder of your application's assets folder. If the constructor fails to load the key, a signal is emitted so that the UI layer can display an error message to the user. WeatherClient::m_apiKeyPath and Weather::m_keyTag are string constants that respectively identify the full path to the key file and the corresponding JSON tag.

> **Note** You will need an API key for the Weather Underground service. You will therefore have to create a developer account at www.wunderground.com/weather/api. You will then be able to generate a new key that you can set in the wunderground.json file located in your project's assets/apikey folder.

REST Service Request

A service request is handled by the WeatherClient::cityWeather(QString city) member function (see Listing 7-15).

Listing 7-15. WeatherClient::cityWeather(QString city)

```
void WeatherClient::cityWeather(QString city) {
        QString urlString("http://api.wunderground.com/api/");
        urlString.append(WeatherClient::m_apiKey);
        urlString.append("/conditions/q/");

        urlString.append(city);
        urlString.append(".json");

        QNetworkRequest request;
        request.setUrl(QUrl(urlString));

        QNetworkReply* reply = this->m_networkManager->get(request);
        bool result = connect(reply, SIGNAL(finished()), this,
                        SLOT(onCityRequestFinished()));
        Q_ASSERT(result);
        this->m_networkReplies.append(reply);
}
```

The WeatherClient::cityWeather(QString city) function dynamically creates a GET request URL by concatenating the city parameter and the API key previously loaded in the class constructor (the constructed URL will have the following structure: http://api.wunderground.com/api/<api key>/conditions/q/<city>.json). As soon as the GET request has been submitted, you will have to connect the QNetworkReply's finished() signal to the WeatherClient's onCityRequestFinished() slot. Finally, when the request has completed, WeatherClient::onCityRequestFinished() will be called (see Listing 7-16).

Working with the Returned JSON

Before actually looking at how the returned JSON document is parsed by the `WeatherClient::onCityRequestFinished()` slot, let us quickly study the structure of the document returned by the Weather Underground service. As a matter of fact, you can conveniently use your browser to perform HTTP requests and study the responses returned by the service. For example, you can use the following URL to retrieve the weather conditions for Los Angeles: `http://api.wunderground.com/api/<key_value>/conditions/q/Los Angeles, CA.json`.

The corresponding JSON structure is shown in Listing 7-16 (note that in order to save some page space, I have removed the `JSON` elements that we will not need to parse or use in our code).

Listing 7-16. Wunderground JSON Response, Single City

```
{
    "response": {
        "version": "0.1",
        "termsofService": "http://www.wunderground.com/weather/api/d/terms.html",
        "features": {
            "conditions": 1
        }
    },
    "current_observation": {
        "display_location": {
            "full":"Los Angeles, CA",
            "city":"Los Angeles",
            "state":"CA",
            "state_name":"California",
            "country":"US",
            "latitude":"33.97457886",
            "longitude":"-118.24745941",
        },
        "observation_time":"Last Updated on October 7, 3:58 AM PDT",
        "weather":"Clear",
        "temperature_string":"63.1 F (17.3 C)",
        "icon_url":"http://icons-ak.wxug.com/i/c/k/nt_clear.gif"
    }
}
```

Remembering what I previously told you about parsing `JSON` documents with a JsonDataAccess object, you can see the following:

- From the structure of the document shown in Listing 7-11, the root object is a `QVariantMap`. Supposing that `result` is the `QVariant` variable obtained with the call to `JsonDataAccess::load()`, the root object is therefore obtained with a call to `result.toMap()`.
- One level down, the `current_observation` object contained in the root object is retrieved using `result.toMap()["current_observation"].toMap()`.
- Similarly, the latitude attribute is retrieved by chaining method calls as follows: `result.toMap()["current_observation"].toMap()["display_location"].toMap()["latitude"].toString()`.

Once you get the hang of chaining the method calls, you will see that you can parse arbitrarily complex JSON structures.

There will be cases where the JSON response will return a list of cities instead of a single observation (this will happen when the city request matches multiple values). For example, if your request URL is http://api.wunderground.com/api/<key_value>/conditions/q/Los Angeles.json (note the missing state specification), the returned JSON document will be given in Listing 7-17.

Listing 7-17. Wunderground JSON Response, Multiple Results

```
{
    "response": {
        "version": "0.1",
        "termsofService": "http://www.wunderground.com/weather/api/d/terms.html",
        "features": {
          "conditions": 1
        },
        "results": [
        {
            "name": "Los Angeles",
            "city": "Los Angeles",
            "state": "CA",
            "country": "US",
            "country_iso3166":"US",
            "country_name":"USA",
            "zmw": "90001.1.99999",
            "l": "/q/zmw:90001.1.99999"
        },
        {
            "name": "Los Angeles",
            "city": "Los Angeles",
            "state": "",
            "country": "CH",
            "country_iso3166":"CL",
            "country_name":"Chile",
            "zmw": "00000.10.85703",
            "l": "/q/zmw:00000.10.85703"
         },
         {
             "name": "Los Angeles",
             "city": "Los Angeles",
             "state": "",
             "cojuntry": "PH",
             "country_iso3166":"PH",
             "country_name":"Philippines",
             "zmw": "00000.31.98752",
                       "l": "/q/zmw:00000.31.98752"
          }
          ]
    }
}
```

Here again, it is quite easy to retrieve the list of cities using the following call chain:

```
result.toMap()["response"].toMap()["results"].toList()
```

And finally, if the request contains an error, the returned JSON document will be similar to Listing 7-18.

Listing 7-18. JSON Response with Error

```
{
  "response": {
      "version":"0.1",
      "termsofService":"http://www.wunderground.com/weather/api/d/terms.html",
      "features": {
      },
      "error": {
          "type": "keynotfound",
          "description": "this key does not exist"
      }
  }
}
```

In other words, you can check for the presence of an error object inside the response in order to make sure that your request was handled correctly by the service (the presence of the error object would be given by the following call chain: `result.toMap()["response"].toMap()contains("error")`).

Now that you have a basic understanding of the JSON document structure, you can see how the service response is parsed in the `WeatherClient::OnCityRequestFinished()` slot (see Listing 7-19).

Listing 7-19. WeatherClient::onCityRequestFinished()

```
void WeatherClient::onCityRequestFinished() {
    QNetworkReply* reply = static_cast<QNetworkReply*>(QObject::sender());
    if (!reply->error()) {
        JsonDataAccess jda;
        QVariant response = jda.load(reply);
        QVariantMap map = response.toMap();
        if (map.contains(WeatherClient::m_currentObservationTag)) {
            this->updateCityInfo(map);
        } else { // else 1
            if (map[WeatherClient::m_responseTag].toMap().contains(
            WeatherClient::m_errorTag)) {
                emit error(map[WeatherClient::m_responseTag]
                           .toMap()[WeatherClient::m_errorTag]
                           .toMap()[WeatherClient::m_descriptionTag].toString());
            } else { // else 2
                m_cities.clear();
                QVariantList results = map[WeatherClient::m_responseTag]
                                       .toMap()[WeatherClient::m_resultsTag].toList();
                for (int i = 0; i < results.length(); i++) {
                    QVariantMap city = results[i].toMap();
```

```
                    if (city[WeatherClient::m_countryTag].toString()
                        == WeatherClient::m_USATag) {
                            m_cities.append(city[WeatherClient::m_nameTag].toString()
                            + ", "+ city[WeatherClient::m_stateTag].toString());
                    } else { // else 3
                        m_cities.append(city[WeatherClient::m_nameTag].toString() + ", "
                        + city[WeatherClient::m_countryNameTag].toString());
                    } // else 3
                } // for
                    emit multipleCitiesFound(m_cities);
            } // else 2
        } // else 1
    }
    m_networkReplies.removeOne(reply);
    reply->deleteLater();
    emit finished();
}
```

Here is a quick description of the code:

1. You will need to handle three cases in the response: a response can either contain the current weather conditions for a city, a list of cities, or an error object. Before even handling the response, we first need to check that the request was handled correctly and that there are no errors in the `QNetworkReply` object.
2. We then proceed by parsing the `JSON` response.
3. If the `JSON` result contains a `current_observation` object, we handle it immediately with a call to `WeatherClient::updateCityInfo(const QVariantMap& map)`.
4. Otherwise, we check if a service error has occurred. If this is the case, we emit the `error` signal with the corresponding error message.
5. If there are no errors, then multiples cities have been returned by the request. In this case, we populate the `m_citiesList QStringList` and emit the `multipleCitiesFound(m_citiesList)` signal, which will be handled in QML.
6. Finally, we schedule the `QNetworkReply` object for deletion and emit the `finished()` signal.

The `WeatherService::updateCityInfo(const QVariantMap& map)` method (used in Listing 7-20) is straightforward and is used for updating the `m_cityInfo` member variable (which is accessible as the `cityinfo` property from QML).

Listing 7-20. WeatherClient::updateCityInfo()

```
void WeatherClient::updateCityInfo(const QVariantMap& data) {
    QVariantMap currentObservation =
    data[WeatherClient::m_currentObservationTag].toMap();
    m_cityInfo->setCity(currentObservation[WeatherClient::m_displayLocationTag]
                        .toMap()[WeatherClient::m_cityTag].toString());
    m_cityInfo->setState(currentObservation[WeatherClient::m_displayLocationTag]
                         .toMap()[WeatherClient::m_stateNameTag].toString());

    m_cityInfo->setWeather(currentObservation[WeatherClient::m_weatherTag].toString());

    m_cityInfo->setTemperature(currentObservation[WeatherClient::m_temperatureTag]
                               .toString());

    m_cityInfo->setCoordinates(currentObservation[WeatherClient::m_displayLocationTag]
                               .toMap()[WeatherClient::m_latitudeTag].toString(),
                               currentObservation[WeatherClient::m_displayLocationTag]
                               .toMap()[WeatherClient::m_longitudeTag].toString(),
                               currentObservation[WeatherClient::m_iconUrlTag].toString());

    m_cityInfo->setLastObservation(currentObservation[WeatherClient::m_observationTimeTag]
                                   .toString());
}
```

CityInfo

Listing 7-21 gives you the `CityInfo` class definition.

Listing 7-21. CityInfo Class Definition

```
#ifndef CITY_H_
#define CITY_H_
#include <QObject>
#include <bb/cascades/Image>
#include <QNetworkAccessManager>

class CityInfo : public QObject {
    Q_OBJECT
    Q_PROPERTY(QString city READ city NOTIFY cityChanged)
    Q_PROPERTY(QString state READ state NOTIFY stateChanged)
    Q_PROPERTY(QString latitude READ latitude NOTIFY latitudeChanged)
    Q_PROPERTY(QString longitude READ longitude NOTIFY longitudeChanged)
    Q_PROPERTY(QString weather READ weather NOTIFY weatherChanged)
    Q_PROPERTY(QVariant weatherIcon READ weatherIcon NOTIFY weatherIconChanged)
    Q_PROPERTY(QString temperature READ temperature NOTIFY temperatureChanged)
    Q_PROPERTY(QString lastObservation READ lastObservation NOTIFY lastObservationChanged)

public:
    CityInfo(QObject* parent = 0);
    virtual ~CityInfo();
```

```
    void setCoordinates(const QString& latitude, const QString& longitude,
                        const QString& weatherIconUrl);

        // accessors.
    void setCity(const QString& city);
    QString city() const;

    void setState(const QString& state);
    QString state() const;

    void setLatitude(const QString& latitude);
    QString latitude() const;

    void setLongitude(const QString& longitude);
    QString longitude() const;

    void setWeather(const QString& weather);
    QString weather() const;

    void setTemperature(const QString& temperature);
    QString temperature() const;

    void setLastObservation(const QString& lastUpdated);
    QString lastObservation() const;

signals:
    void cityChanged();
    void stateChanged();
    void latitudeChanged();
    void longitudeChanged();
    void coordinatesChanged(const QString& latitude, const QString& longitude,
                                const  QString& markerUrl);
    void weatherChanged();
    void weatherIconChanged();
    void temperatureChanged();
    void lastObservationChanged();

private slots:
        void onWeatherIconRequestFinished();

private:
    QVariant weatherIcon()const;

    void setWeatherIconUrl(const QString& iconUrl);
    void downloadWeatherIcon(const QString& iconUrl);

    QNetworkAccessManager* m_networkManager;
    QString m_city;
    QString m_state;
    QString m_latitude;
    QString m_longitude;
    QString m_temperature;
```

```
    QString m_lastObservation;
    QString m_weather;
    QString m_weatherIconUrl;
    bb::cascades::Image m_weatherIcon;
};
```

Note that the properties declared in the class definition are the ones used by the QML `City` control bindings (see Listing 7-11). Also, the `Notify` signals are required for updating the QML bindings when the C++ properties change.

If you look at Figure 7-3, you will notice that a small icon is used for representing the current weather conditions. The Weather Underground service provides a `URL` pointing to a downloadable image representing the current conditions (see the `icon_url` element in the `JSON` response in Listing 7-16). The CityInfo class therefore uses the URL to download the icon and display it in QML as an `ImageView`. Listings 7-22 and 7-23 provide the code for downloading the image.

Listing 7-22. CityInfo::downloadWeatherIcon

```
void CityInfo::downloadWeatherIcon(const QString& iconUrl) {
    QNetworkRequest request;
    request.setUrl(QUrl(iconUrl));

    QNetworkReply* reply = this->m_networkManager->get(request);
    bool result = connect(reply, SIGNAL(finished()), this,
                          SLOT(onWeatherIconRequestFinished()));
    Q_ASSERT(result);
}
```

You should be quite familiar by now with the code shown in Listing 7-22. An `HTTP` request for downloading the image is created and submitted to the network access manager. The interesting part of the code is located in Listing 7-23, which handles the `HTTP` response.

Listing 7-23. CityInfo::onWeatherIconRequestFinished

```
void CityInfo::onWeatherIconRequestFinished() {
    QNetworkReply* reply = static_cast<QNetworkReply*>(QObject::sender());
    if (reply) {
        if (reply->error() == QNetworkReply::NoError) {
            QByteArray data = reply->readAll();
            m_weatherIcon = bb::cascades::Image(bb::utility::ImageConverter::decode(data));
            emit weatherIconChanged();
        }
        reply->deleteLater();
    }
}
```

The code essentially builds a `bb::cascades::Image` from the returned data using a `bb::utility::ImageConverter` class, and updates the `m_weatherIcon` member variable. Note that we also need to emit the `weatherIconChanged` signal, which will in turn notify the declarative engine to update the QML binding for the `City.weatherImage` property.

The last piece of the puzzle is to access the Image object as a QVariant from QML using the weatherIcon property (see Listing 7-24).

Listing 7-24. CityInfo::onWeatherIcon()

```
QVariant CityInfo::weatherIcon() const {
    return QVariant::fromValue(m_weatherIcon);
}
```

GoogleMapClient

The GoogleMapClient class generates a static map using the coordinates returned by the Weather Underground service. Here again, the class encapsulates the map generation functionality and exclusively uses properties and signals to communicate with the QML layer. When the GoogleMapClient::setCoordinates() slot is called, a new request to the Google Maps service is sent. If you look at the WeatherMap control's onCreationCompleted slot, you will notice that the CityInfo::coordinatesChanged() signal is connected to the GoogleMapClient::setCoordinates() slot (see Listing 7-12) (in other words, the GoogleMapClient()::setCoordinates() slot will be called each time the CityInfo object's coordinates are updated).

Listing 7-25 shows you the GoogleMapClient::setCoordinates() slot implementation.

Listing 7-25. CityInfo::setCoordinates()

```
void GoogleMapClient::setCoordinates(const QString& latitude,
    const QString& longitude, const QString& markerUrl) {
    if((m_latitude == latitude) &&
       (m_longitude == longitude) &&
       (m_markerUrl == markerUrl)) return;
    m_latitude = latitude;
    m_longitude = longitude;
    m_markerUrl = markerUrl;
    this->createMap();
}
```

Finally, the GoogleMapClient::setCoordinates() method internally calls the GoogleMapClient::createMap() method, which is responsible for building the network request to the Google Maps service (see Listing 7-26).

Listing 7-26. GoogleMapClient::createMap()

```
void GoogleMapClient::createMap() {
    QNetworkRequest request;
    request.setUrl(QUrl(this->buildUrlString()));
    QNetworkReply* reply = this->m_networkManager->get(request);
    bool result = connect(reply, SIGNAL(finished()), this, SLOT(onMapReady()));
    Q_ASSERT(result);
}
```

I am going to omit the code for handling the HTTP response, which is done in GoogleMapClient::onMapReady(), because it is very similar to WeatherClient::onWeatherIconRequesFinished() (shown in Listing 7-23). (In retrospect, we could have designed a common base class implementing the image download logic. This is something you could try to refactor.)

The request URL is built with a call to GoogleMapClient::buildUrlString() (see Listing 7-27).

Listing 7-27. GoogleMapClient::buildUrlString()

```
QString GoogleMapClient::buildUrlString() {
    QString cityMapUrl("http://maps.googleapis.com/maps/api/staticmap?center=");
    cityMapUrl.append(m_latitude);
    cityMapUrl.append(",");
    cityMapUrl.append(m_longitude);
    cityMapUrl.append("&");
    cityMapUrl.append("zoom=7&size=640x640&sensor=false&");
    cityMapUrl.append("maptype=hybrid&");
    cityMapUrl.append("markers=");
    cityMapUrl.append("icon:");
    cityMapUrl.append(m_markerUrl);
    cityMapUrl.append("|");
    cityMapUrl.append(m_latitude);
    cityMapUrl.append(",");
    cityMapUrl.append(m_longitude);
    cityMapUrl.append("|");
    cityMapUrl.append("scale=2");
    return cityMapUrl;
}
```

The code shown in Listing 7-27 essentially creates a new request for a map centered on the m_latitude and m_longitude coordinates. The marker parameter for indicating the coordinates is defined as the URL of the icon returned by the Weather Underground service. (If you specify an image URL as a marker, Google Maps will add it as a marker on your map. By default, when no markers are specified, Google will use its own for the coordinates). This illustrates how you can combine, in practice, multiple services in your own app (we could say that we have built a mashable app).

If you are interested in finding out more about the Google static maps API, you can refer to the following URL: https://developers.google.com/maps/documentation/staticmaps.

ApplicationUI

As usual for Cascades applications, the application delegate ties everything together and provides you the access point for the WeatherClient and CityInfo instances (note that the delegate itself is set as a QML document context property; see Listings 7-28 and 7-29).

Listing 7-28. ApplicationUI Definition

```
class ApplicationUI : public QObject
{
    Q_OBJECT
    Q_PROPERTY(WeatherClient* weather READ weatherClient CONSTANT)
```

```
public:
    ApplicationUI(bb::cascades::Application *app);
    virtual ~ApplicationUI() { }

private:
    WeatherClient* weatherClient();
    WeatherClient* m_weatherClient;
};
```

Listing 7-29. ApplicationUI Constructor

```
ApplicationUI::ApplicationUI(bb::cascades::Application *app) :
                QObject(app), m_weatherClient(new WeatherClient(this)) {

        // Create scene document from main.qml asset, the parent is set
        // to ensure the document gets destroyed properly at shut down.
        QmlDocument *qml = QmlDocument::create("asset:///main.qml").parent(this);
        qml->documentContext()->setContextProperty("_app", this);

        // Create root object for the UI
        AbstractPane *root = qml->createRootObject<AbstractPane>();

        // Set created root object as the application scene
        app->setScene(root);
}
```

Finally, to make the `WeatherClient`, `CityInfo` and `GoogleMapClient` classes available as new QML types, you need to register them with the QML type system. This is done in the application's `main` function (see Listing 7-30).

Listing 7-30. main.cpp

```
Q_DECL_EXPORT int main(int argc, char **argv)
{
    qmlRegisterType<CityInfo>("ludin.utils", 1, 0, "CityInfo");
    qmlRegisterType<WeatherClient>("ludin.utils", 1, 0, "WeatherClient");
    qmlRegisterType<GoogleMapClient>("ludin.utils", 1, 0, "GoogleMapClient");

    Application app(argc, argv);

    // Create the Application UI object, this is where the main.qml file
    // is loaded and the application scene is set.
    new ApplicationUI(&app);

    // Enter the application main event loop.
    return Application::exec();
}
```

The first two calls to qmlRegisterType() are required because you are using CityInfo and WeatherClient as properties accessible from QML. The last call is required so that you can define the GoogleMapClient class as an attached object in the WeatherMap control. (You also need to add the import ludin.utils 1.0 statement at the start of your QML document; see the WeatherMap control in Listing 7-12.)

Summary

This chapter provided an overview of the BlackBerry 10 networking classes based on the QtNetwork module. The networking classes are completely generic, but this chapter showed you how to use them for the HTTP protocol. QNetworkManager plays the role of the grand dispatcher to submit network requests and handle responses. The class supports the usual HTTP verbs (GET, PUT, and POST), which makes it a breeze to use with restful services. An HTTP request is encapsulated by a QNetworkRequest instance and the response can be handled using a corresponding QNetworkReply instance. Networking is completely asynchronous, thus ensuring that UI thread is not blocked during an HTTP request. Finally, it should be emphasized that the networking classes are reentrant, meaning that you can call them multiple times from a single thread without corrupting their state.