Chapter 1

Getting Started

This chapter will show you how to set up your BlackBerry 10 development environment and deploy your first application on the BlackBerry 10 simulator and on a physical device. You will also get a broad perspective of the Cascades programming model, as well as its most essential features. In setting up your environment, I will walk you through the following steps:

- Getting your code signing keys and generating debug tokens.
- Using the Momentics IDE to create your first Cascades project.
- Building and deploying your application on a simulator and a physical device.

Cascades Programming Model

BlackBerry 10 is a major mobile operating system overhaul. It's the third release built on top of the extremely reliable QNX operating system, which is used in critical applications ranging from medical devices to nuclear power plants. QNX is also *POSIX compliant*, meaning that if you're familiar with a UNIX programming API, you will feel just at home with the operating system's calls. Another big advantage of building BlackBerry 10 on top of a POSIX system is the availability of a myriad of open-source libraries that you can include in your own projects.

A key feature of BlackBerry 10 is that it is built using a multilayered architecture where QNX is the backbone providing essential services such as multithreading, memory management, and security, to name a few (see Figure 1-1). The layer on top of QNX includes the BlackBerry Platform Services (BPS) as well as several modules from the Qt framework.

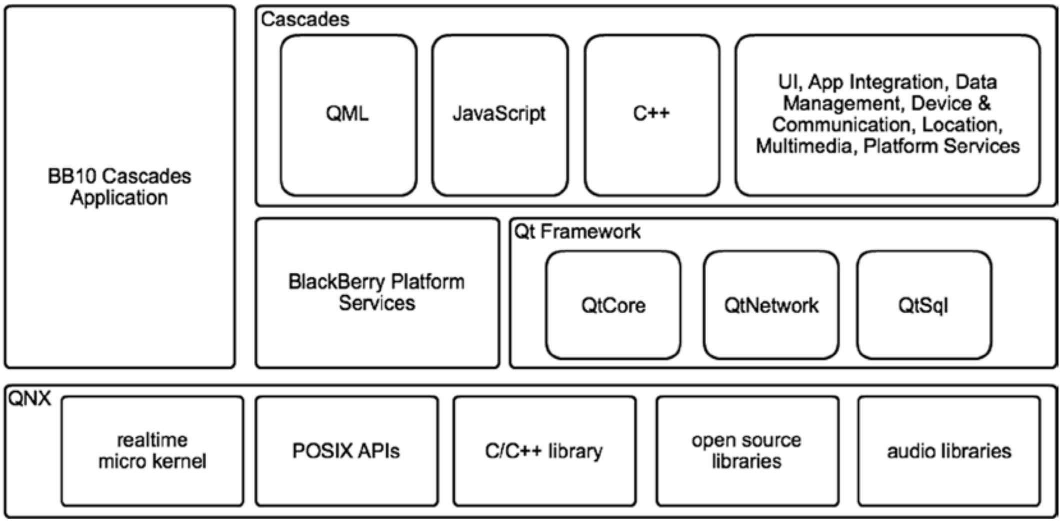

BB10 Cascades Application
Cascades
QML
JavaScript
C++
UI, App Integration, Data Management, Device & Communication, Location, Multimedia, Platform Services
BlackBerry Platform Services
Qt Framework
QtCore
QtNetwork
QtSql
QNX
realtime micro kernel
POSIX APIs
C/C++ library
open source libraries
audio libraries

***Figure 1-1.** BlackBerry 10 platform*

BPS is an API written in C, giving low-level access to the BlackBerry 10 device. It's mostly used when you need to write high-performance applications such as games that require the most effective way of accessing the hardware. BPS is not the main subject of this book. I will nevertheless give you examples of how to use it, but I will mostly concentrate on the higher-level modules built on top of BPS.

Qt is a C++ framework providing an abstraction layer to the lower-level POSIX APIs. It also adds many classes and components essential to C++ programming. The following modules from the Qt framework have been ported to the BlackBerry 10 platform and can be used in your own applications:

- *QtCore*: Provides the core framework elements for building C++ applications. In particular, QtCore defines the Qt object system, an event handling mechanism called *signals and slots*, memory management, and collection classes, to name a few.
- *QtNetwork*: Provides APIs for building networked applications. In particular, for HTTP applications, it provides the `QNetworkAccessManager` class.
- *QtSql*: Includes drivers and data access logic to relational databases.
- *QtXml*: Includes SAX and DOM parsers for handling XML documents.

The Qt modules mostly provide non-GUI functionality for your application. To build rich native applications with an engaging UI, you need to rely on the Cascades layer of the BlackBerry 10 architecture. In fact, Cascades is much more than a GUI framework; it also includes the following nonexhaustive list of services and APIs:

- *User interface*: Provides the core components for building rich native user interfaces using QML/JavaScript, C++, or a mix of all three technologies.
- *Application integration*: APIs that integrate platform applications and functionality such as e-mail and calendar into your own apps.

- *Data management*: High-level APIs abstracting data sources and data models. The supported data formats include SQL, XML, and JSON.
- *Communication*: APIs for enabling your apps to communicate with other devices by using, for example, Bluetooth, Wi-Fi, and NFC.
- *Location*: APIs for using maps and managing location services in your application.
- *Multimedia*: APIs for accessing the camera, audio player, and video player in your apps.
- *Platform*: Additional APIs for managing platform notifications and home screen functions.

When developing native applications, you will notice that there is some overlap between the functionality provided by Cascades and the underlying modules. At first this might seem confusing but you should keep in mind that Cascades often provides a richer and easier-to-use API. Therefore, as a good rule of thumb, always try to implement a functionality with the Cascades API first, and if it is not possible, use the underlying Qt or BPS modules. Networking is a good example where you will use the QtNetwork module essentially.

QML

When building user interfaces with Cascades, you can proceed in two distinct ways: you can either write imperative code in C++ or create your UI declaratively with the Qt Modeling Language (QML). Most examples in this book use the latter approach for the following reasons:

- Thanks to the Cascades Builder tool, you get immediate feedback on the way your UI will look in QML.
- When it comes to designing UIs, writing C++ code can quickly become unmanageable, especially if you consider many nested components. In contrast, QML keeps the code much more tractable.
- Once you get the hang of QML, it is way faster to create a polished UI within a few minutes than in C++.
- Behind the scenes, you are still using C++ objects exposed to QML by Cascades. QML simply makes your life easier during the entire application development life cycle by avoiding numerous compile-build-deploy cycles until you get the UI right.
- QML is a much friendlier language than C++ for people with a programming background in JavaScript. You will therefore have a greater chance of sharing your UI designs with other members of your team if they are written in QML.

To illustrate the previous points, let's design a very simple UI using both approaches: one UI design in QML and another one in C++. As shown in Figure 1-2, the UI isn't very fancy; it's simply a text field stacked on top of a slider. Whenever the slider moves, the text field is updated with the slider's new position.

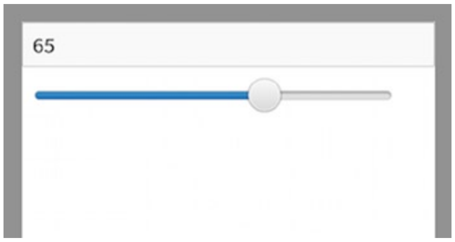


Figure 1-2. Stacked TextField and Slider

Listing 1-1 shows the QML markup version.

Listing 1-1. main.qml

```
import bb.cascades 1.0
Page {
    Container {
        TextField {
           id: texfield
        }
        Slider{
            id: slider
            fromValue: 0
            toValue: 100
            onImmediateValueChanged: {
                texfield.text = Math.round(immediateValue)
            }
        }
    }
}
```

The equivalent C++ version of the code for creating the same UI is given in Listings 1-2 and 1-3.

> Don't worry if you have never programmed in C++, we will cover the basics in Chapter 3. As a matter of fact, you will also see in Chapter 2 that you can build relatively complex Cascades applications using QML/JavaScript only, without ever writing a single line of C++ code.

Listing 1-2. applicationui.hpp

```
class ApplicationUI : public QObject
{
    Q_OBJECT
public:
    ApplicationUI(bb::cascades::Application *app);
    virtual ~ApplicationUI() { }
public slots:
    void onImmediateValueChanged(float value);

};
```

Listing 1-3. applicationui.cpp

```
ApplicationUI::ApplicationUI(bb::cascades::Application *app) : QObject(app) {
   Page *page = new Page();

   Container *contentContainer = new Container();
   contentContainer->setLayout(StackLayout::create());

   TextField* textfield = TextField::create();
   Textfield->setObjectName("textfield");
   Slider* slider = Slider::create();
   slider->setFromValue(0);
   slider->setToValue(100);

   contentContainer->add(textfield);
   contentContainer->add(slider);

   QObject::connect(slider, SIGNAL(immediateValueChanged(float)), this,
                SLOT(onImmediateValueChanged (float)));

   page->setContent(contentContainer);
   app->setScene(page);
}

void ApplicationUI::onImmediateValueChanged(float value) {
    value = round(value);
    QString stringValue = QString::number(value);
    Application* app = static_cast<Application*>(this->parent());
    TextField* textField = app->scene()->findChild<TextField*>("textfield");
    textField->setText(stringValue);
}
```

`ApplicationUI` is the "application delegate" in charge of creating the user interface and wiring together the application's controls' event handling. You have to provide this class and it is instantiated during the application bootstrap process.

As you can see, the declarative way of building the UI in QML is very concise compared to the imperative C++ approach. This is also because Cascades takes care of a lot of the plumbing work for you behind the scenes when you're using QML.

Signals and Slots

In Cascades terminology, event handling is done using signals and slots, which are basically a loosely coupled notification mechanism between controls. Whenever something interesting happens to a control, such as a state change, a predefined signal is emitted for notifying that change. If you're interested in receiving that notification, then you have to specify some application logic in JavaScript or C++, which will be called in the corresponding Cascades predefined signal handler. Signals and slots are part of the QtCore module. The Cascades framework uses them in order to build a high-level event handling mechanism. This section will expand on the topic in order to give you a firm grip on the way signals and slots work. As noted previously, the most important property of signals is their ability to let you bind objects together without them knowing about each other.

Signals and Slots in QML

For a given predefined `signal` signal, Cascades also provides a corresponding predefined `onSignal` handler (which is also called equivalently a slot). You can write JavaScript code in your QML document to tell Cascades what to do when the handler is triggered and how the control should respond to the signal. For example, in order to handle the slider's position updates, Cascades defines a predefined `onImmediateValueChanged` signal handler called when the slider emits the `immediateValueChanged` signal. In Listing 1-1, the predefined handler will execute the `texfield.text = Math.round(immediateValue)` JavaScript code in order to update the textfield. You will also notice that the JavaScript code references an `immediateValue` parameter. Signals usually include extra parameters that provide additional information about them. In QML, they are implicitly available to the JavaScript execution context and you can use them in order to retrieve further information about the change that just occurred.

You can refer to the Cascades API reference found at `http://developer.blackberry.com/cascades/reference/user_interface.html` for a list of all predefined signals and corresponding slots organized by GUI control. Look under the core controls section.

Signals and Slots in C++

Looking at Listing 1-2, you will notice that I've used the `slots:` annotation to declare an `onImmediateValueChanged(float value)` slot in the application delegate class. In Listing 1-3, I've connected the slider's `onImmediateValueChanged(float value)` to the application delegate's `onImmediateValueChanged(float value)` slot using the `QObject::connect(source, SIGNAL(signal), destination, SLOT(slot))` method.

> The `Q_OBJECT`, `signals:` and `slots:` "annotations" are Qt extensions to the C++ language.

Signals and slots are implemented in Qt using the following constructs:

- A class must inherit from `QObject`.
- You must add the `Q_OBJECT` macro at the beginning of the class definition. The `Q_OBJECT` macro marks the class as managed by the Meta Object Compiler (MOC). During compilation, the MOC generates additional code for the class in a file called `moc_classname.cpp`, which adds support for signals and slots, metaprogramming, and other features for runtime introspection. Note that the entire process is completely transparent and you don't need to worry about it during compilation.

If you intend on extending the class, you must also repeat the `Q_OBJECT` macro in all of its subclasses.

- You must declare the class signals using the `signals:` annotation.
- You must declare the class slots using the `slots:` annotation.
- You must define the class slots as regular member functions.
- Finally, you must wire signals and slots using `QObject::connect()`.

As an example, let us consider the case of a temperature sensor. We would like to build a system where we can chart and log temperature readings over time. We would also want to decouple the system by separating the charting logic from the temperature logging. A very simplified design can be implemented using three classes (see Figure 1-3). The `TempSensor` class is responsible for the temperature readings through the `setTemp(float newValue)` function, which could be triggered by a hardware interrupt. The function would then update `TempSensor`'s internal state, and then emit a `tempChanged(float)` signal. The `TempChart` and `TempLogger` classes would respectively handle the signal with a corresponding `onTempChanged(float)` slot.

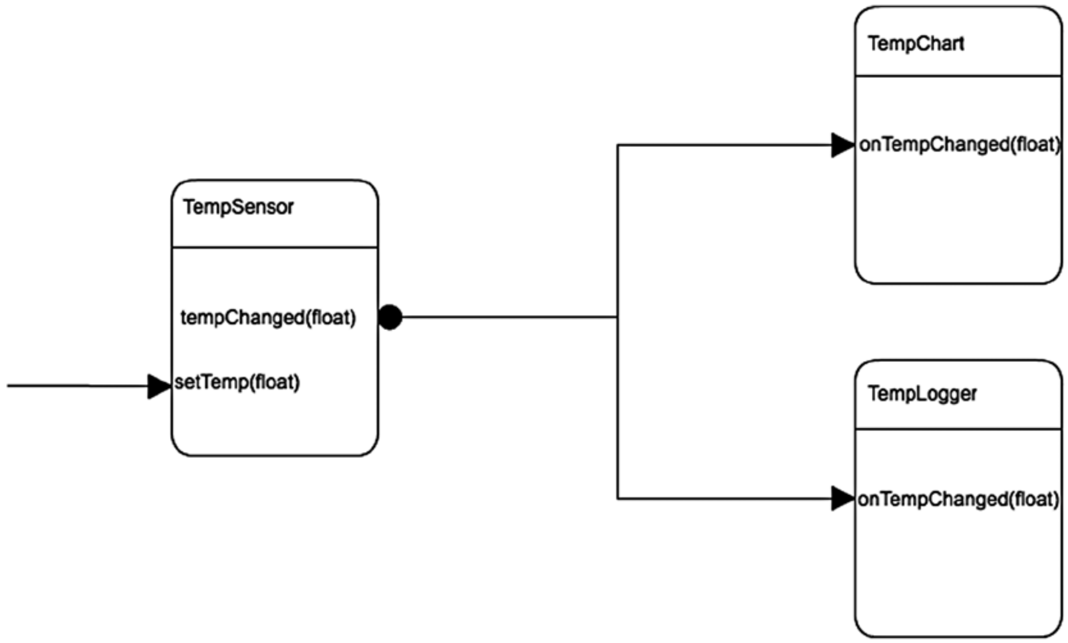


***Figure 1-3.** Sensor system*

The C++ implementation is given in Listings 1-4 and 1-5.

Listing 1-4. TempSensor.hpp

```
#include <QObject>

class TempSensor : public QObject{
Q_OBJECT
public:
    TempSensor(QObject* parent=0) : QObject(parent), m_value(0) {};
    virtual ~TempSensor(){};

    void setTemp(float newValue){
        if(m_value == newValue) return;
        m_value = newValue;
        emit(tempChanged(m_value);
    }

signals:
    void tempChanged(float)

private:
    float m_value;
};

#include <QObject>


class TempChart : public QObject{
Q_OBJECT
public:
    TempChart(QObject* parent=0) : QObject(parent){};
public slots:
    void onTempChanged(float value){
        // do charting
    }
};

#include <QObject>

class TempLogger : public QObject{
Q_OBJECT
public:
    TempLogger(QObject* parent=0) : QObject(parent){};

public slots:
    void onTempChanger(float value){
        // do logging
    }
};
```

Listing 1-5. main.cpp

```
#include "TempSensor.hpp"
int main(){
    TempSensor sensor;
    TempLogger logger;
    TempChart chart;

    QObject::connect(sensor, SIGNAL(tempChanged(float)), logger, SLOT(onTempChanged(float)));
    QObject::connect(sensor, SIGNAL(tempChanged(float)), chart, SLOT(onTempChanged(float)));

    // do temperature readings here.
}
```

Here are a few things to keep in mind when implementing signals and slots:

- Signals are triggered in your code using the emit `signalName()` syntax (see Listing 1-4).
- Signals must always have a void return value. In other words, you can't get a return value from a signal once it has been emitted.
- As illustrated in the previous example, one signal can be connected to many slots. When the signal is emitted, the slots are called one after the other.
- The opposite is also true; many signals can be connected to the same slot.
- You can also connect a signal to another signal. When the first signal is emitted, the second one is also emitted.
- Slots are normal member functions. You can call them directly if you wish. They can also be virtual functions if you wish.
- The signature of a signal must match the signature of the receiving slot. A slot can also have a shorter signature than the signal (in this case the slot drops the extra arguments).

Meta-Object System

Qt extends C++ with a meta-object system in order to introduce runtime introspection features that would not be available with a statically compiled language such as C++. Behind the scenes, Qt uses the meta-object compiler (MOC) to generate the extra C++ plumbing code for the functions declared by the `Q_OBJECT` macro and for the class signals. Finally, the `QObject::connect` function uses the MOC-generated introspection functions to wire signals and slots together. When building Cascades applications, the MOC is called transparently by the build system.

Cascades Application Bootstrap Process

The entry point for all Cascades applications is the `main` function shown in Listing 1-6.

Listing 1-6. main.cpp

```
#include <bb/cascades/Application>
#include <QLocale>
#include <QTranslator>
#include "applicationui.hpp"

#include <Qt/qdeclarativedebug.h>

using namespace bb::cascades;

Q_DECL_EXPORT int main(int argc, char **argv)
{
    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 step in `main` is to create an instance of a `bb::cascades::Application` class, which provides the application's run loop, and all the boilerplate functionality required by a Cascades application. At this point, you will have a "bare bones" Cascades app but the run loop has not kicked in yet. To further customize the application, the following properties of the `bb::Cascades::Application` instance have to be specified:

- *Scene property*: Specifies the instance of `bb::cascades::AbstractPane` to use as the scene for the application's main window. A scene is basically a layout of controls which will be displayed in the application's main window.
- *Cover property*: Specifies the instance of `bb::cascades::AbstractCover` to be used when the application is in cover mode.
- *Menu property*: An instance of a `bb::cascades::Menu` accessible by the user with a swipe from the top of the screen.

In practice, you will not update the `bb::cascades::Application`'s properties directly in the `main` function but instead rely on an application delegate object, which will take care of loading or creating the main scene and wiring all the events using signals and slots. You've already seen an implementation of an application delegate in Listing 1-2 and Listing 1-3 given by the `ApplicationUI` class. In Listing 1-3, we customized the application delegate in order to build the scene graph using C++. Listing 1-7 shows the default version generated by the Momentics IDE's New BlackBerry Project wizard (more on installing your development environment later in the chapter).

Listing 1-7. applicationui.cpp

```
ApplicationUI::ApplicationUI(bb::cascades::Application *app) :
        QObject(app)
{
    // prepare the localization. Code omitted
    // 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);

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

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

I've removed the code related to localization in order to concentrate on the scene graph creation logic. Here an instance of a `bb::cascades::QmlDocument` is created by reading the `main.qml` QML file containing the declarative UI description. This is the same QML you will design using the Cascades Builder tool.

Finally, once the application delegate has been initialized, the application's main event loop kicks in through a call to `bb::cascades::Application::exec()`.

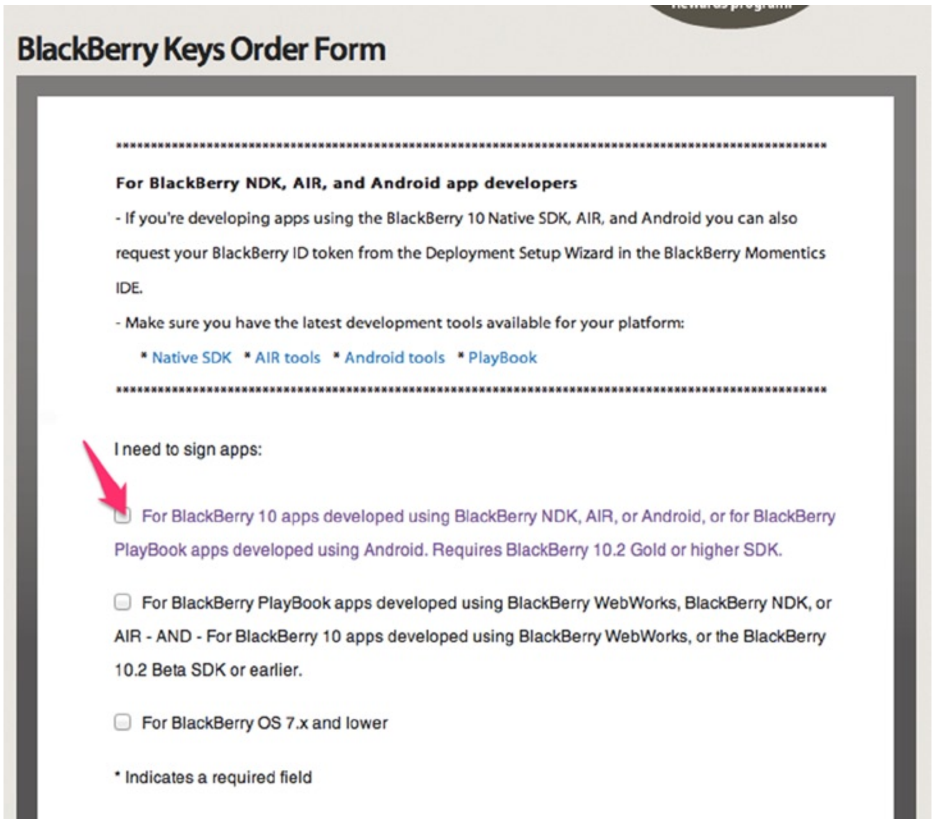
Parent-Child Ownership

If you take a close look at Listing 1-3, you will notice that I haven't released the objects allocated with the `new` operator at any point in the code. This might seem as a memory leak but it's not. Cascades widgets are organized in a parent-child relationship that also handles object ownership and memory management. In the case shown in Listing 1-3, the root parent of the entire object hierarchy is the `bb::cascades::Application app` object. The memory associated with the child controls will be released when this object is deleted by the runtime. I will cover memory management in detail in Chapter 3, but for the moment you can safely assume that there are no memory leaks in Listing 1-3.

Native SDK Setup

To build Cascades applications, you need to set up the native SDK using the following steps:

1. Download and install the latest version of the Momentics IDE from `http://developer.blackberry.com/native/downloads` (the page will also provide you with a link to the latest BlackBerry 10 simulator). You can either download the simulator directly or let Momentics handle the download at a later stage when you configure a simulator target.
2. Request a BlackBerry ID from `http://blackberryid.blackberry.com`. You will need your BlackBerry ID to create a BlackBerry ID *token*, which is used in turn for generating debug tokens (debug tokens are deployed on a BlackBerry device during development and enable your device to run development code). Note that you don't need a debug token for the simulator.
3. As soon as you have created your BlackBerry ID, go to `https://www.blackberry.com/SignedKeys` in order to generate a BlackBerry ID token. Select the first option and sign in with your BlackBerry ID (see Figure 1-4).



BlackBerry Keys Order Form

****

For BlackBerry NDK, AIR, and Android app developers

- If you're developing apps using the BlackBerry 10 Native SDK, AIR, and Android you can also request your BlackBerry ID token from the Deployment Setup Wizard in the BlackBerry Momentics IDE.

- Make sure you have the latest development tools available for your platform:

* Native SDK * AIR tools * Android tools * PlayBook

****

I need to sign apps:

☐ For BlackBerry 10 apps developed using BlackBerry NDK, AIR, or Android, or for BlackBerry PlayBook apps developed using Android. Requires BlackBerry 10.2 Gold or higher SDK.

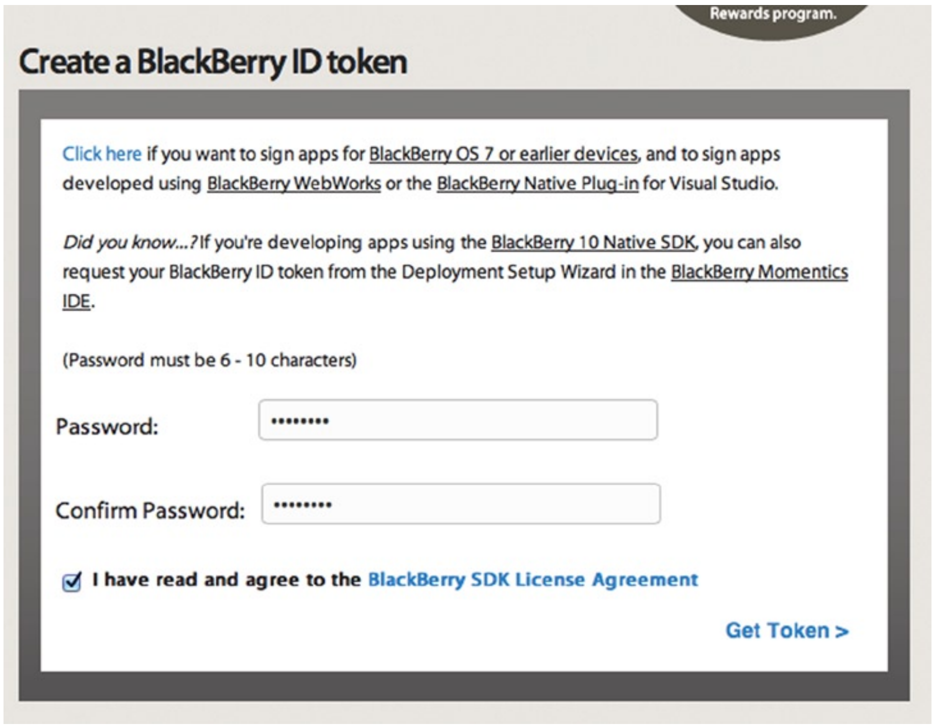
☐ For BlackBerry PlayBook apps developed using BlackBerry WebWorks, BlackBerry NDK, or AIR - AND - For BlackBerry 10 apps developed using BlackBerry WebWorks, or the BlackBerry 10.2 Beta SDK or earlier.

☐ For BlackBerry OS 7.x and lower

* Indicates a required field

***Figure 1-4.** BlackBerry keys order form*

4. After having signed in, you will be redirected to another page for generating your BlackBerry ID token. Enter a password for the token, accept the license agreement, and click Get Token (see Figure 1-5).


Rewards program.

Create a BlackBerry ID token

Click here if you want to sign apps for BlackBerry OS 7 or earlier devices, and to sign apps developed using BlackBerry WebWorks or the BlackBerry Native Plug-in for Visual Studio.

Did you know...? If you're developing apps using the BlackBerry 10 Native SDK, you can also request your BlackBerry ID token from the Deployment Setup Wizard in the BlackBerry Momentics IDE.

(Password must be 6 - 10 characters)

Password: •••••••••

Confirm Password: •••••••••

✓ I have read and agree to the BlackBerry SDK License Agreement

Get Token >

***Figure 1-5.** BlackBerry ID token*

5. The token will be generated and downloaded as a file called `bbidtoken.csk`. Depending on your development platform, you will have to put the file in one of the following locations:
   a. Windows XP: `C:\Documents and Settings\Application Data\Research in Motion\`
   b. Windows Vista, Windows 7, and Windows 8: `C:\Users\AppData\Local\Research in Motion\`
   c. Mac OS X: `~/Library/Research in Motion`

Momentics IDE

To create Cascades applications, you will use the Momentics IDE, which essentially adds extra plug-ins and tools to a standard Eclipse distribution (if you've already used Eclipse in the past for Java or Android development, you will be right at home; otherwise, don't worry—this section will guide you through the IDE). This section explains how a Cascades project is organized in Momentics and reviews the most important features of the IDE that you will be using frequently. First start by creating a new Cascades project using the following steps:

1. Launch the Momentics IDE and choose File ➤ New ➤ BlackBerry Project… This will start the New BlackBerry Project wizard shown in Figure 1-6.

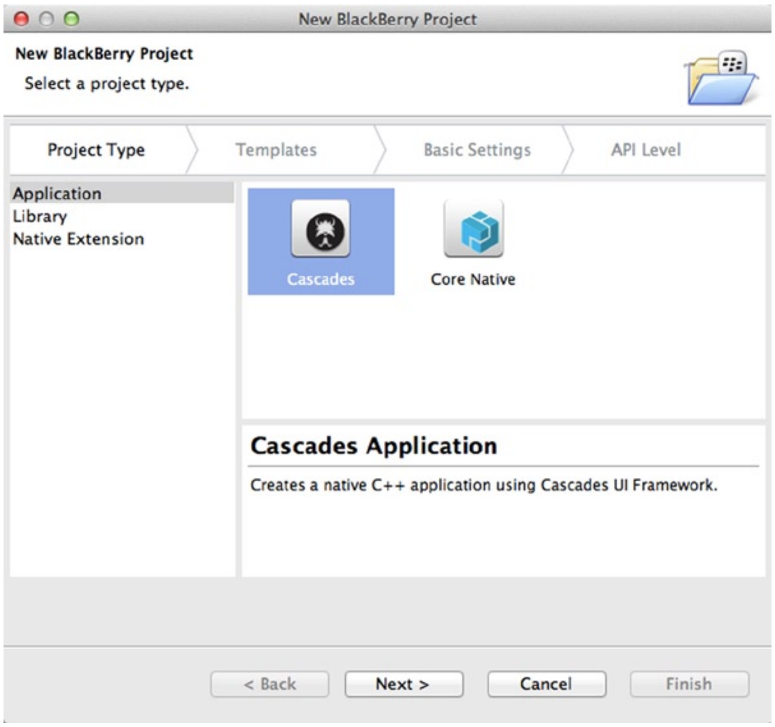


***Figure 1-6.** BlackBerry 10 Platform*

2. Select Cascades as the project type and click Next.
3. Select Standard Empty Project from the templates page and click Next.
4. On the Basics Settings page, change your project's name from the default CascadesProject to HelloCascades, and then click Next. Don't change any of the other default settings.
5. Keep the default settings on the last wizard page API Level and click Finish.
6. If you're not in the QML Editing perspective, a prompt will appear, asking you if you want to switch to it. Click Yes.

Workspace

Momentics stores your projects in a *workspace*, which is essentially a collection of projects located in the same directory on your file system. Once you've finished creating the HelloCascades project, your workspace should look similar to Figure 1-7.

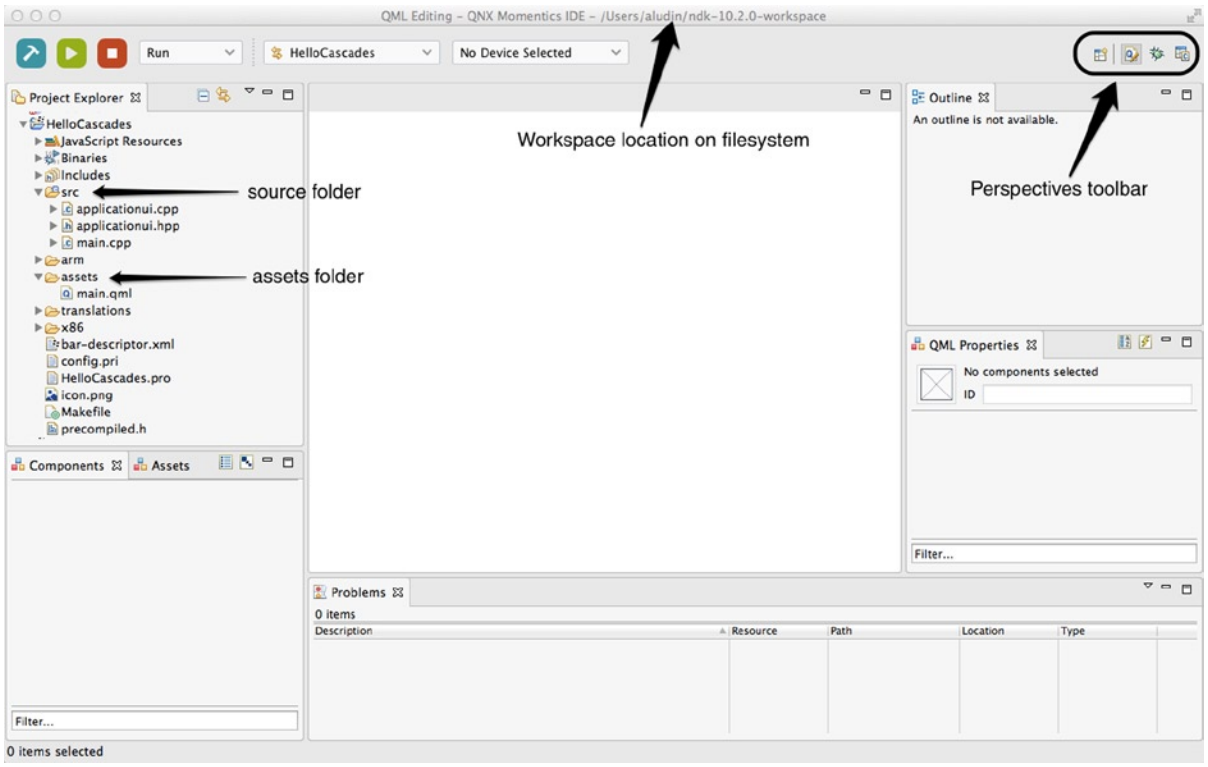


Figure 1-7. Momentics workspace

Perspectives

A *perspective* is a task-oriented collection of views and editors. When designing Cascades applications, you will mostly use the QML Editing, C/C++, and Debug perspectives. You can easily switch from one perspective to another using the perspectives toolbar or the Window ➤ Open Perspective navigation menu. Some views, such as the Project Explorer, will appear in multiple perspectives.

In the Project Explorer view, the `src` subfolder contains the following C++ source files:

- `main.cpp`: Defines the application entry point `main`.
- `applicationui.hpp` and `application.cpp`: You will find the wizard-generated application delegate declaration and definition.

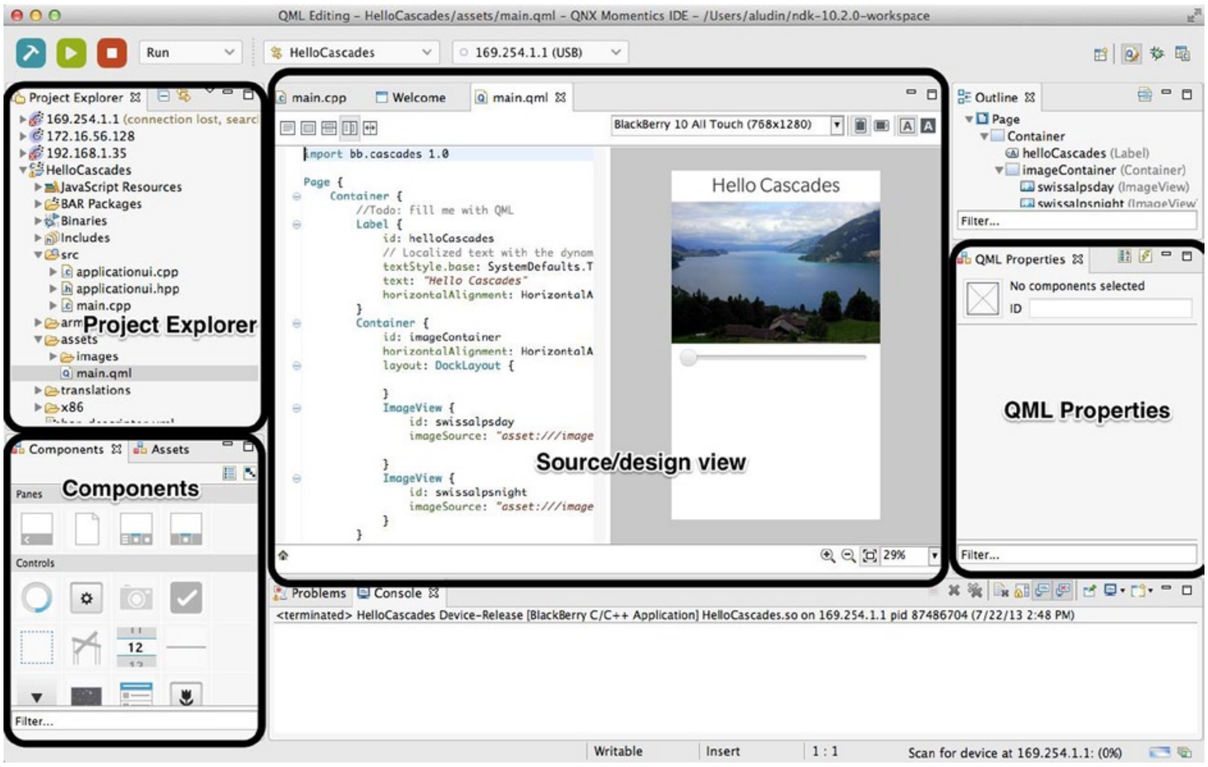
You've already seen simplified versions of these files in the examples in Listing 1-7. For the moment, you can simply ignore them. The `assets` subfolder contains the `main.qml` defining your application's UI.

Let's spice up the default version of the app generated by the Cascades wizard.

1. Create a new folder called `images` under the `assets` folder of your project (see Figure 1-5).
2. Copy the `swissalpsday.png` and `swissalpsnight.png` from the book's resources in your project's `images` folder.

The source code for this book is located in the `https://github.com/aludin/BB10Apress` GitHub repository and at `www.apress.com/9781430261575`. You can either clone the repository or download a compressed Zip copy. As you read along, you can import the projects in turn in Momentics (in Momentics, select File ➤ Import Existing Projects into Workspace and select the root directory of a project located under the `BB10Apress` folder).

3. Open the `main.qml` file by double-clicking it in Project Explorer. Make sure you're in the QML editing perspective by switching to it using the perspectives toolbar located in the upper-right corner of the Momentics IDE. The QML editing or Cascades Builder perspective is organized around four important views (see Figure 1-8):



```
import bb.cascades 1.0

Page {
    Container {
        //Todo: fill me with QML
        Label {
            id: helloCascades
            // Localized text with the dynam
            textStyle.base: SystemDefaults.T
            text: "Hello Cascades"
            horizontalAlignment: HorizontalA
        }
        Container {
            id: imageContainer
            horizontalAlignment: HorizontalA
            layout: DockLayout {

            }
            ImageView {
                id: swissalpsday
                imageSource: "asset:///image

            }
            ImageView {
                id: swissalpsnight
                imageSource: "asset:///image
            }
        }
```

```
▸ 169.254.1.1 (connection lost, searc
▸ 172.16.56.128
▸ 192.168.1.35
▾ HelloCascades
  ▸ JavaScript Resources
  ▸ BAR Packages
  ▸ Binaries
  ▸ Includes
  ▾ src
    ▸ applicationui.cpp
    ▸ applicationui.hpp
    ▸ main.cpp
  ▸ arm
  ▾ assets
    ▸ images
      main.qml
  ▸ translations
  ▸ x86
```

```
<terminated> HelloCascades Device-Release [BlackBerry C/C++ Application] HelloCascades.so on 169.254.1.1 pid 87486704 (7/22/13 2:48 PM)
```

Figure 1-8. Momentics IDE, QML perspective

- The Project Explorer shows you all the resources available in your project, including source folders, asset folders, and targets.
- The Components view located on the lower-left section of the screen displays core Cascades controls that you can drag and drop in the Source view located at the center of your screen.

- The QML Properties view is displayed on the right side of the screen. You can use this view by selecting a QML element in the Source view.
- The main design area is located in the middle of your screen. You can switch between source only, design only, and source-design split modes.

4. In the Source view, remove the `text: qsTr(Hello World) + Retranslate.onLocaleOrLanguageChanged` property from the Label control.
5. Select the Label in the Source view by double-clicking it, and then update the QML Properties view by doing the following:
   - Add “helloCascades” in the id field.
   - Add “Hello Cascades” in the text field.
   - Scroll down until you reach the Horizontal Alignment property of the label and change it to `Center`.

`main.qml` should now look like Listing 1-8.

Listing 1-8. main.qml

```
import bb.cascades 1.0

Page {
    Container {
        //Todo: fill me with QML
        Label {
            id: helloCascades
            // Localized text with the dynamic translation and locale updates support
            textStyle.base: SystemDefaults.TextStyles.BigText
            text: "Hello Cascades"
            horizontalAlignment: HorizontalAlignment.Center
        }
    }
}
```

6. Drag a Container control from the Components view and drop it under the label’s closing brace in the Source view.
7. Double-click the second Container control:
   - Change the id to imageContainer.
   - Change the Horizontal Alignment property to Center.
   - Change the Layout property to DockLayout.
8. Drag an ImageView control from the Components view and drop it after the DockLayout control’s closing brace in the Source view.

9. Select the ImageView control:
   - Change the id property to “swissalpsday”.
   - Click the Image Source button and select the `swissalpsday.png` file in the `assets/images` folder.
10. Add another ImageView control under the previous one in the Source view.
    - Change the id property to “swissalpsnight”.
    - Click the Image Source button and select the `swissalpsnight.png` file in the `assets/images` folder
    - Set the opacity property to 0.
11. Drag a Slider control from the Components view and drop it in the Source view after `imageContainer`’s closing brace. Change the slider Horizontal Alignment to Center.
12. In the Source view, add the following code in the body of the Slider control:

```
   onImmediateValueChanged: {
   swissalpsnight.opacity = immediateValue
}
```

The final version of the QML markup should look like Listing 1-9. If not, try to repeat the previous steps until you reach the same result, or simply update the QML directly in the Source view.

Listing 1-9. main.qml

```
import bb.cascades 1.0

Page {
    Container {
        //Todo: fill me with QML
        Label {
            id: helloCascades
            // Localized text with the dynamic translation and locale updates support
            textStyle.base: SystemDefaults.TextStyles.BigText
            text: "Hello Cascades"
            horizontalAlignment: HorizontalAlignment.Center
        }
        Container {
            id: imageContainer
            horizontalAlignment: HorizontalAlignment.Center
            layout: DockLayout {

            }
            ImageView {
                id: swissalpsday
                imageSource: "asset:///images/swissalpsday.png"
            }
            ImageView {
                id: swissalpsnight
                imageSource: "asset:///images/swissalpsnight.png"
            }
        }
        Slider {
            horizontalAlignment: HorizontalAlignment.Center
            onImmediateValueChanged: {
                swissalpsnight.opacity = immediateValue
            }
        }
    }
}
```

Congratulations! You've just finished designing your first Cascades application!

Build Configurations

There are four build configurations to consider when creating Cascades application:

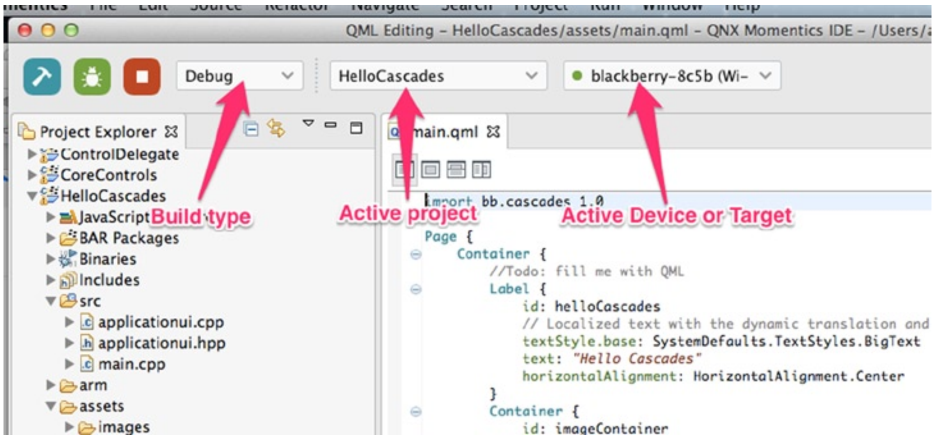
- Simulator debug
- Device debug
- Device profile
- Device release

A build configuration defines a set of rules and settings for building your application for a given processor or target (for example, the "Simulator debug" configuration will build your project with debug symbols enabled for a Simulator target, whereas "Device release" will build a release version of your project for a physical device with an ARM processor). At any point, you can set the active build configuration, as explained in the following paragraph.

To build the project for the simulator, select HelloCascades in Project Explorer, and then set Project ➤ Build Configurations ➤ Set Active ➤ Simulator-Debug from the Momentics main menu. Next, select Project ➤ Build Project. The build starts immediately and the build output is displayed in the Console View.

When the build finishes, a new folder called `x86/o-g` containing the build results will be created under your project's root folder.

Note that another extremely convenient way of selecting a build configuration is by using the BlackBerry Toolbar, as shown in Figure 1-9 (you will also see in the next section how to use the BlackBerry Toolbar to set up targets). To build the project, select Debug for the build type and then click the Hammer button.



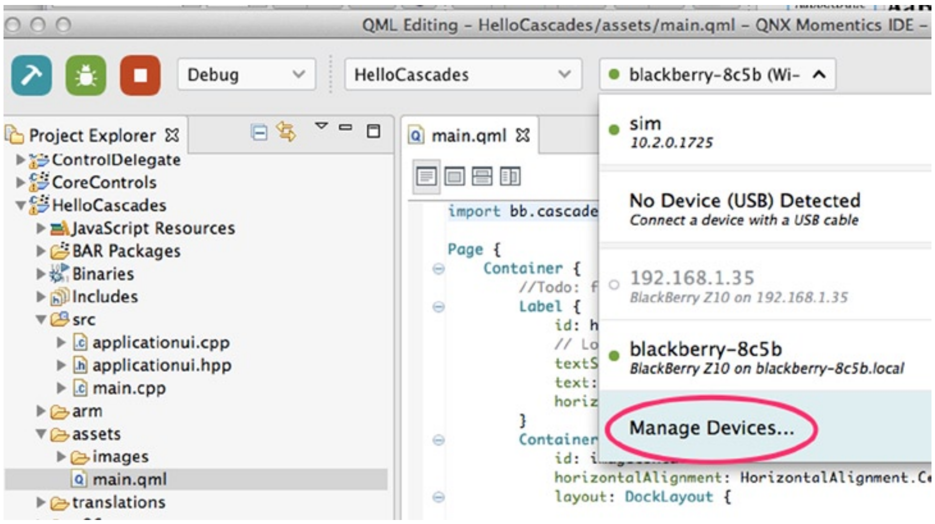
```
import bb.cascades 1.0

Page {
    Container {
        //Todo: fill me with QML
        Label {
            id: helloCascades
            // Localized text with the dynamic translation and
            textStyle.base: SystemDefaults.TextStyles.BigText
            text: "Hello Cascades"
            horizontalAlignment: HorizontalAlignment.Center
        }
        Container {
            id: imageContainer
```

Figure 1-9. BlackBerry Toolbar

Targets

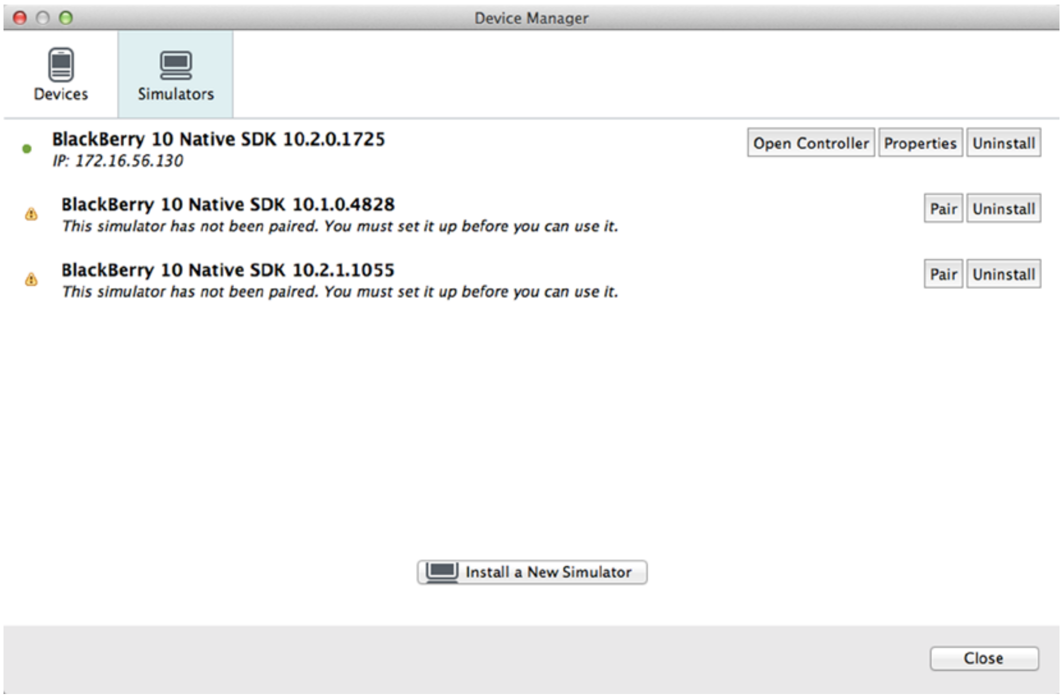
Before testing HelloCascades, you need to define a deployment target. On the BlackBerry Toolbar, select the Manage Devices… option located in the Active Device drop-down (this will display the Device Manager wizard; see Figure 1-10 and Figure 1-11).



```
import bb.cascade

Page {
    Container {
        //Todo: f
        Label {
            id: h
            // Lo
            textS
            text:
            horiz
        }
        Container
            id: i
            horizontalAlignment: HorizontalAlignment.C
            layout: DockLayout {
```

Figure 1-10. Manage devices



BlackBerry 10 Native SDK 10.2.0.1725
IP: 172.16.56.130

BlackBerry 10 Native SDK 10.1.0.4828
This simulator has not been paired. You must set it up before you can use it.

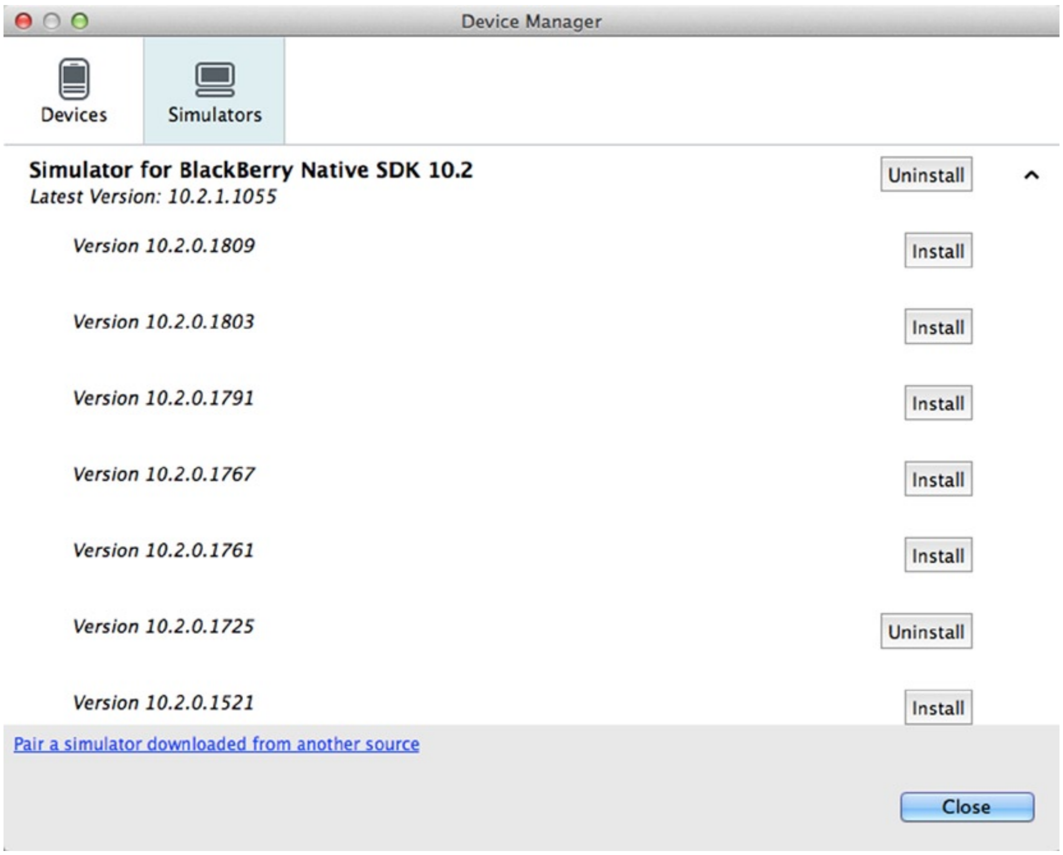
BlackBerry 10 Native SDK 10.2.1.1055
This simulator has not been paired. You must set it up before you can use it.

***Figure 1-11.** Device Manager (figure also shows installed simulators)*

Simulator

To configure a new simulator using the Device Manager wizard, follow these steps:

1. Click Install a New Simulator. Choose the most recent simulator from the list and install it (see Figure 1-12). (Note that if you are developing for a specific API level, you can select a different simulator. I will tell you more about API levels at the end of this Chapter.)



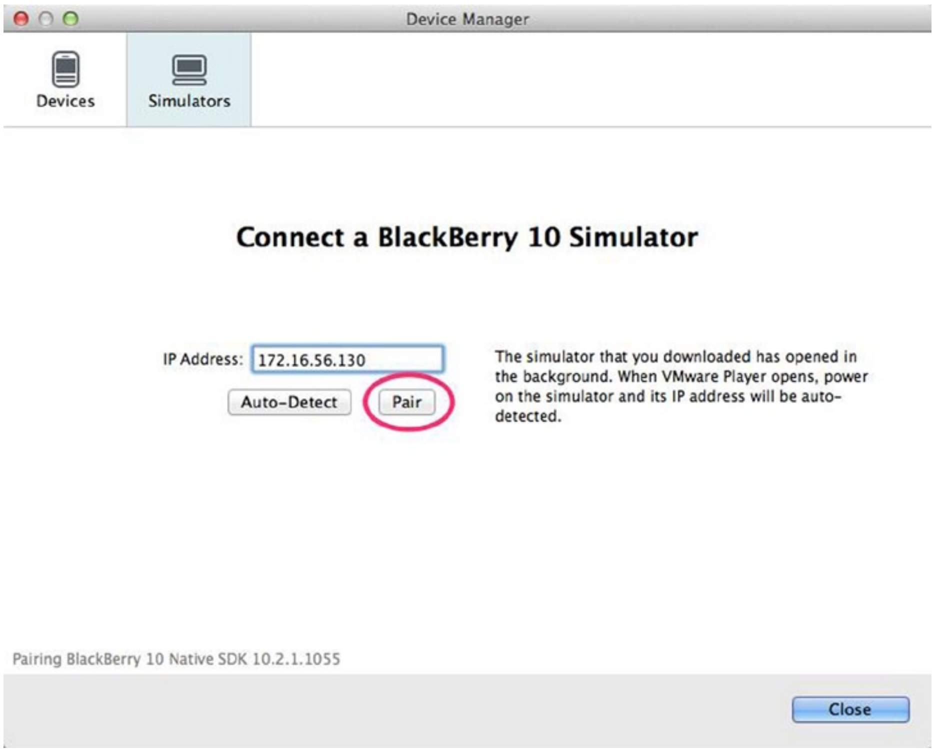
Simulator for BlackBerry Native SDK 10.2 — Uninstall
Latest Version: 10.2.1.1055

Version 10.2.0.1809	Install
Version 10.2.0.1803	Install
Version 10.2.0.1791	Install
Version 10.2.0.1767	Install
Version 10.2.0.1761	Install
Version 10.2.0.1725	Uninstall
Version 10.2.0.1521	Install

Pair a simulator downloaded from another source

***Figure 1-12.** Simulator versions*

2. As soon as you have selected the simulator, the Device Manager wizard will start its download.
3. When the download has completed, the simulator will be launched and the final step will be to pair Momentics with the simulator (see Figure 1-13).



Connect a BlackBerry 10 Simulator

IP Address: 172.16.56.130

The simulator that you downloaded has opened in the background. When VMware Player opens, power on the simulator and its IP address will be auto-detected.

Pairing BlackBerry 10 Native SDK 10.2.1.1055

Figure 1-13. Simulator pairing

4. The simulator will now appear in the Device Manager's list of simulators and you can connect to it (see Figure 1-11). (Note that you might need to restart Momentics for the new simulator to appear in the BlackBerry Toolbar's Active Device list.)

You can now try to launch HelloCascades on the simulator using the green Debug button on the BlackBerry Toolbar (if you haven't built the project previously, click the Hammer button; see Figure 1-9).

Device

Configuring a new physical device for testing purposes is accomplished by pairing the device with Momentics. You will also have to generate a debug token, which will be saved on the device by Momentics. Once again, the BlackBerry Toolbar streamlines the process:

1. Make sure to turn on Development Mode on your device using Settings ➤ Security and Privacy ➤ Development Mode.
2. Connect your device to your computer with the USB cable provided by BlackBerry.

3. Just like for the simulator, launch the Device Manager wizard from the BlackBerry Toolbar. This time, select the Devices tab and click Set Up New BlackBerry 10 Device (see Figure 1-14).

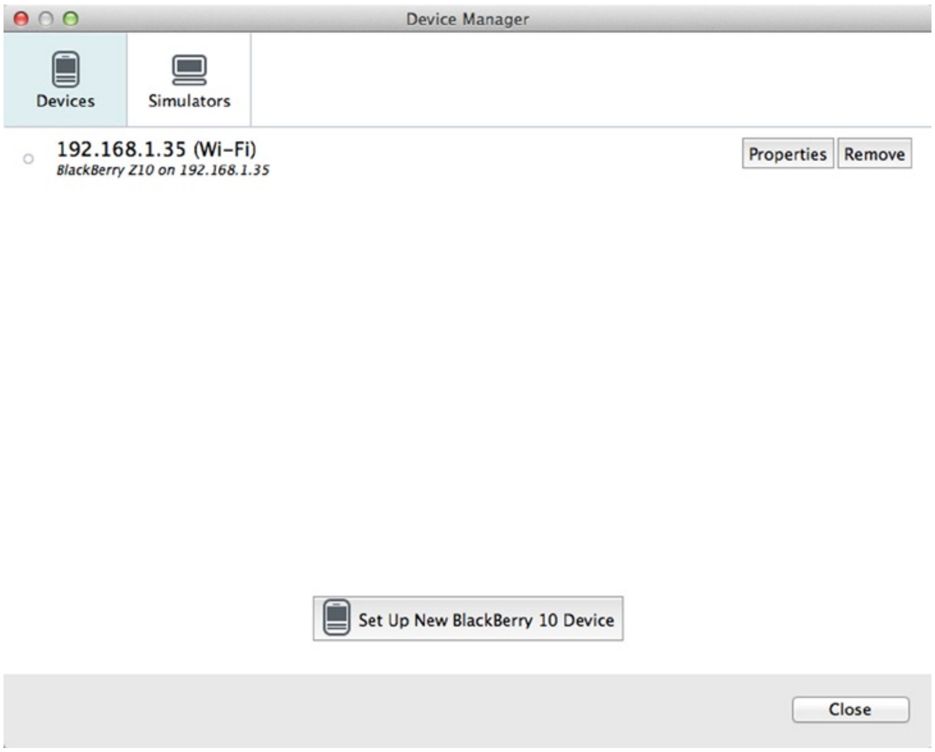
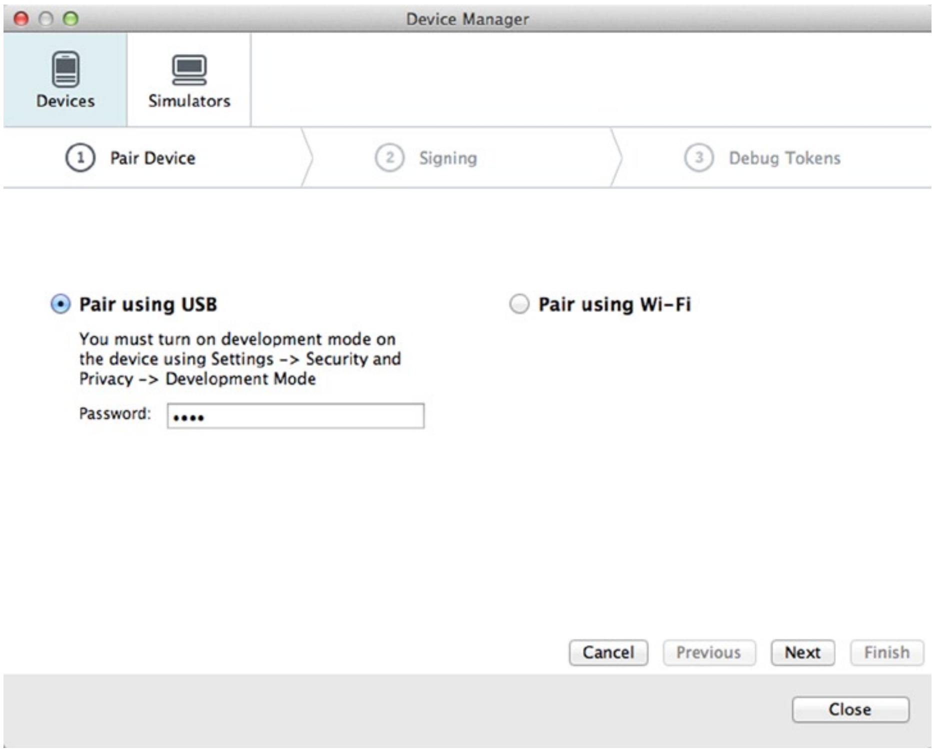


***Figure 1-14.** Set up new BlackBerry 10 device*

4. You will have to pair your device during the first step of the configuration. To pair your device, you can either use the USB cable or a Wi-Fi connection. Select Pair Using USB and then click Next. (Note that if your device is protected by a password, enter it in the password field; see Figure 1-15.)



Pair using USB [✓] Pair using Wi-Fi

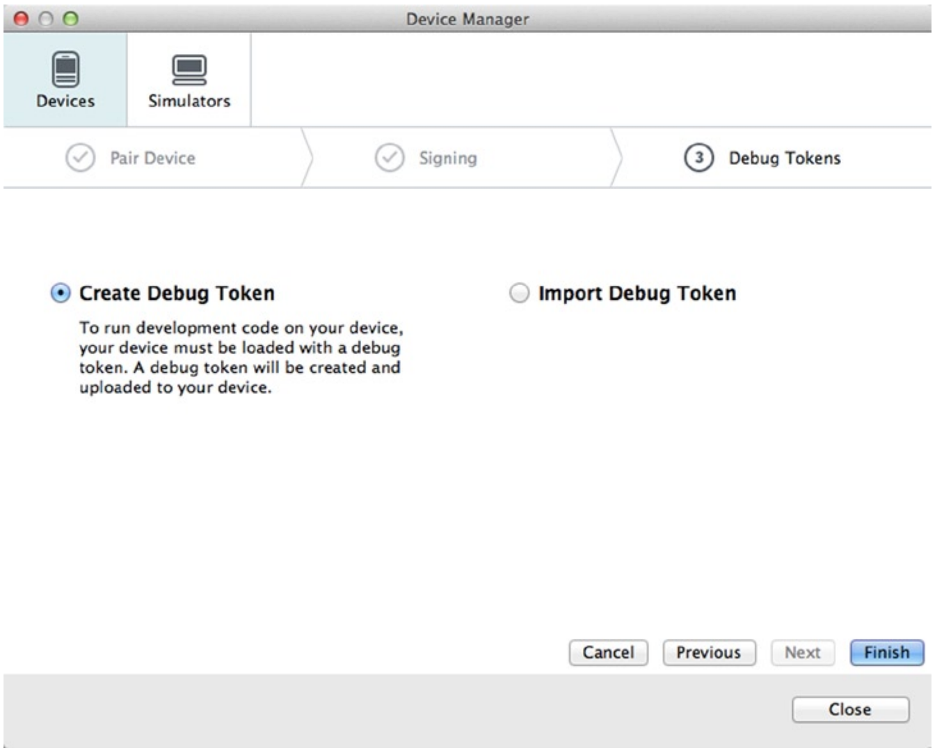
You must turn on development mode on the device using Settings -> Security and Privacy -> Development Mode

Password: ••••

Figure 1-15. Pair device using USB

5. If you have already generated your BlackBerry ID token as explained in the SDK configuration section, the wizard will skip the second step; otherwise, follow the wizard's instructions.

6. On the next wizard page, select Create Debug Token and click Finish. You will finally be asked to provide the password used to create your BlackBerry ID token (see Figure 1-5) before a new debug token is deployed on your device (see Figure 1-16).



Create Debug Token [✓] Import Debug Token

To run development code on your device, your device must be loaded with a debug token. A debug token will be created and uploaded to your device.

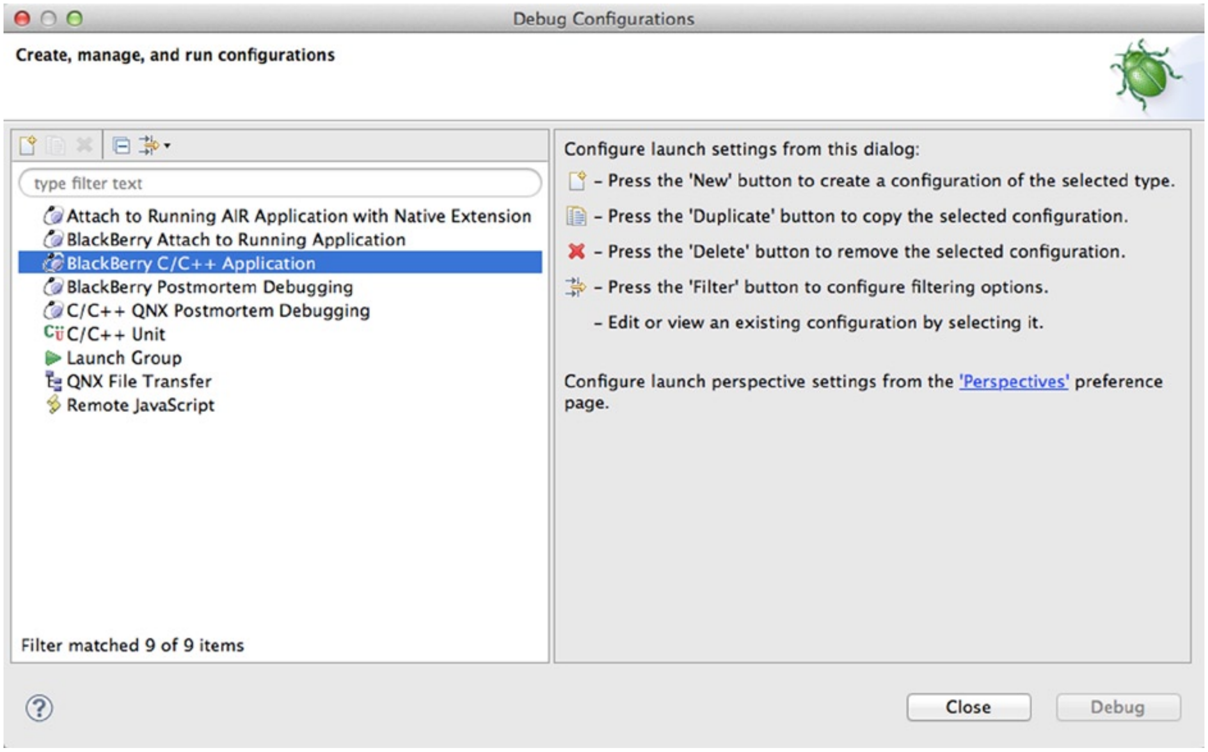
Figure 1-16. Create Debug Token

This time, you can try to launch HelloCascades on the device by selecting it as the Active Device on the BlackBerry Toolbar.

Launch Configurations

The purpose of this section is to explain what's happening behind the scenes when you use the BlackBerry Toolbar, which essentially creates launch configurations for you. A launch configuration is purely an Eclipse concept and not at all specific to Momentics; it associates a build result with a target. You must create it in order to run your application on a simulator or a device. There are two kinds of launch configurations that you can create: the Run Configuration and the Debug Configuration. In this section, I will show you how to create a Debug Configuration for the Simulator target. (The steps for creating a Run Configuration are identical to a Debug Configuration. A Run Configuration will simply launch your application on the target, whereas a Debug Configuration will launch it under Momentics' debugger control.)

1. Select Run ➤ Debug Configurations… from the Momentics main menu to display the Debug Configurations Dialog (see Figure 1-17).



Create, manage, and run configurations

- Attach to Running AIR Application with Native Extension
- BlackBerry Attach to Running Application
- BlackBerry C/C++ Application
- BlackBerry Postmortem Debugging
- C/C++ QNX Postmortem Debugging
- C/C++ Unit
- Launch Group
- QNX File Transfer
- Remote JavaScript

Filter matched 9 of 9 items

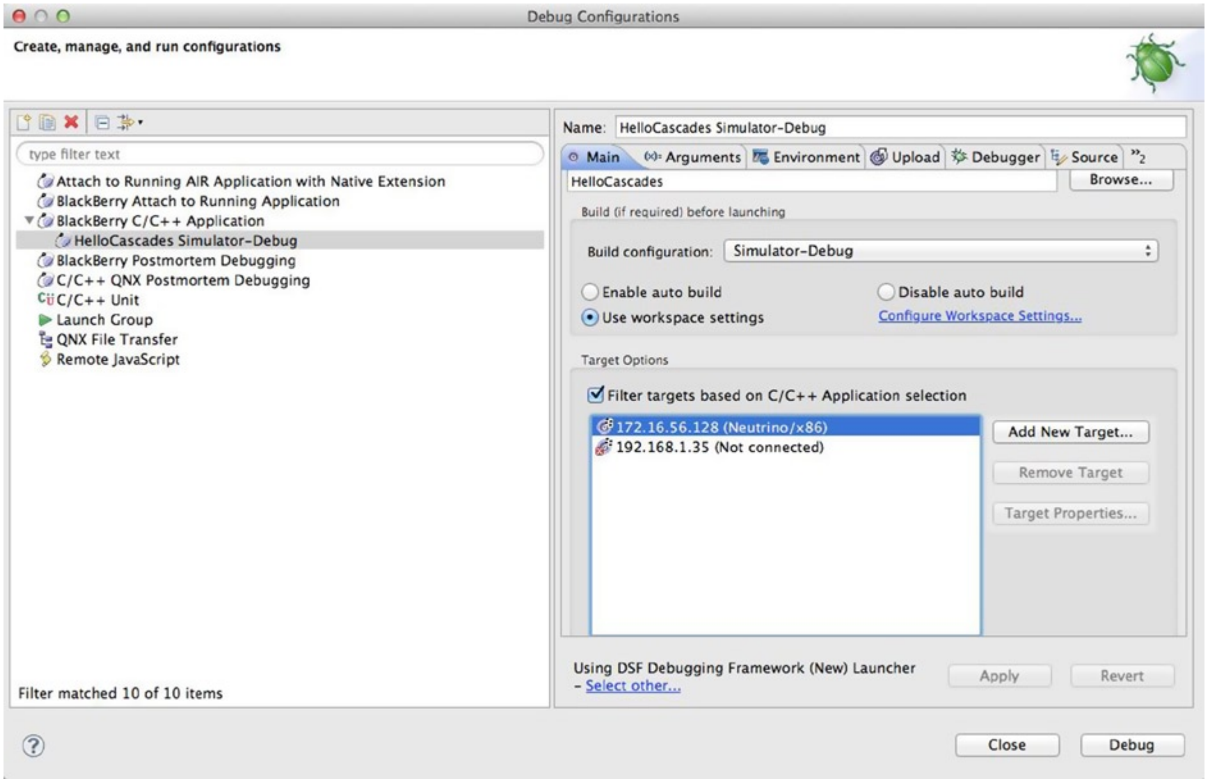
Configure launch settings from this dialog:

- Press the 'New' button to create a configuration of the selected type.
- Press the 'Duplicate' button to copy the selected configuration.
- Press the 'Delete' button to remove the selected configuration.
- Press the 'Filter' button to configure filtering options.
- Edit or view an existing configuration by selecting it.

Configure launch perspective settings from the 'Perspectives' preference page.

***Figure 1-17.** Debug Configurations*

2. Select BlackBerry C/C++ Application from the list and press the New button in the upper-left corner of the dialog box. The settings for the new launch configuration will be displayed (see Figure 1-18).



Debug Configurations

Create, manage, and run configurations

type filter text

- Attach to Running AIR Application with Native Extension
- BlackBerry Attach to Running Application
- BlackBerry C/C++ Application
  - HelloCascades Simulator-Debug
- BlackBerry Postmortem Debugging
- C/C++ QNX Postmortem Debugging
- C/C++ Unit
- Launch Group
- QNX File Transfer
- Remote JavaScript

Filter matched 10 of 10 items

Name: HelloCascades Simulator-Debug

Main | Arguments | Environment | Upload | Debugger | Source | »2

HelloCascades

Build (if required) before launching

Build configuration: Simulator-Debug

Enable auto build | Disable auto build | [✓ Use workspace settings] | Configure Workspace Settings...

Target Options

[✓] Filter targets based on C/C++ Application selection

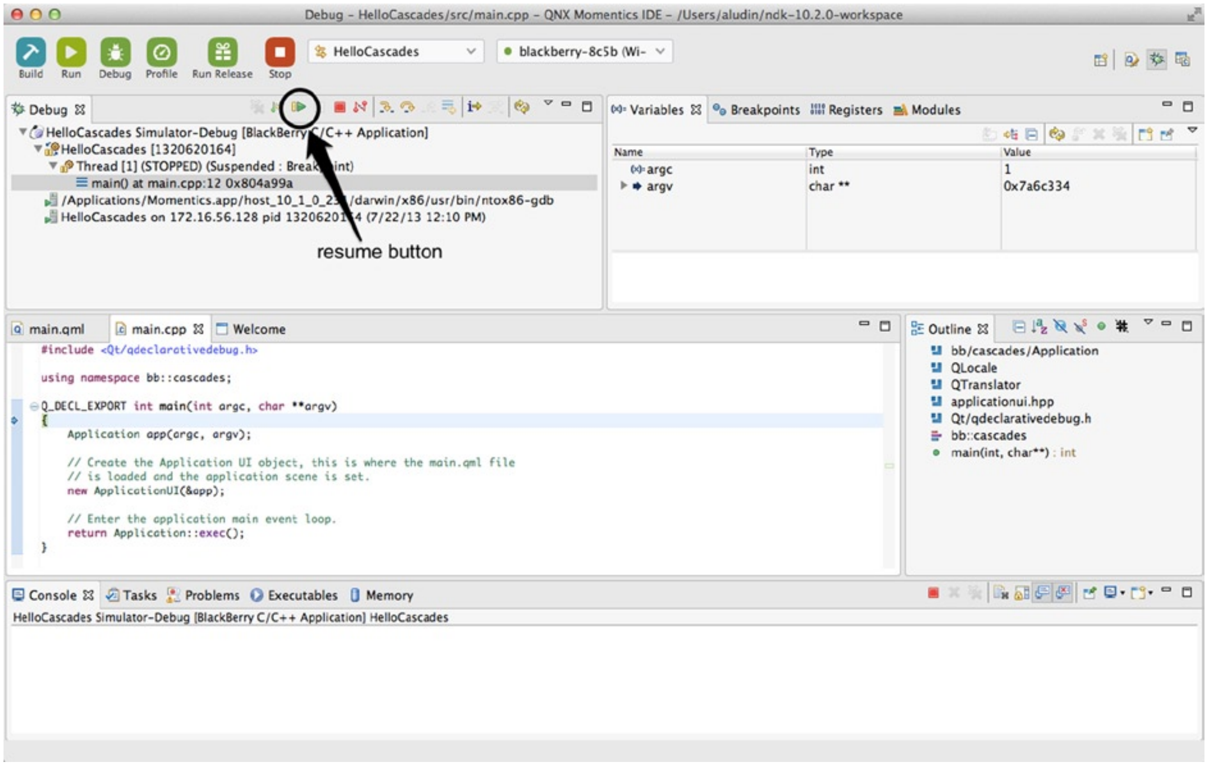
172.16.56.128 (Neutrino/x86) (selected)
192.168.1.35 (Not connected)

Using DSF Debugging Framework (New) Launcher
– Select other...

Figure 1-18. Simulator launch configuration

3. Make sure that the build configuration is Simulator Debug and the selected target is Neutrino/x86, which corresponds to the simulator. Press Apply and then press Debug (note that the simulator name might be different, depending on how you have configured it).

4. HelloCascades will now be launched in debug mode on the simulator. The Momentics IDE will also switch from the QML Editing perspective to the Debug perspective, and the program execution will stop at the beginning of the main function (see Figure 1-19) .



```
HelloCascades Simulator-Debug [BlackBerry C/C++ Application]
  HelloCascades [1320620164]
    Thread [1] (STOPPED) (Suspended : Breakpoint)
      main() at main.cpp:12 0x804a99a
  /Applications/Momentics.app/host_10_1_0_231/darwin/x86/usr/bin/ntox86-gdb
  HelloCascades on 172.16.56.128 pid 1320620164 (7/22/13 12:10 PM)
```

Name	Type	Value
argc	int	1
argv	char **	0x7a6c334

```
#include <Qt/qdeclarativedebug.h>

using namespace bb::cascades;

Q_DECL_EXPORT int main(int argc, char **argv)
{
    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();
}
```

```
bb/cascades/Application
QLocale
QTranslator
applicationui.hpp
Qt/qdeclarativedebug.h
bb::cascades
main(int, char**) : int
```

```
HelloCascades Simulator-Debug [BlackBerry C/C++ Application] HelloCascades
```

***Figure 1-19.** Debug perspective*

5. Press the Resume button to continue the program execution. The Hello Cascades application should now be running on the simulator (see Figure 1-20).

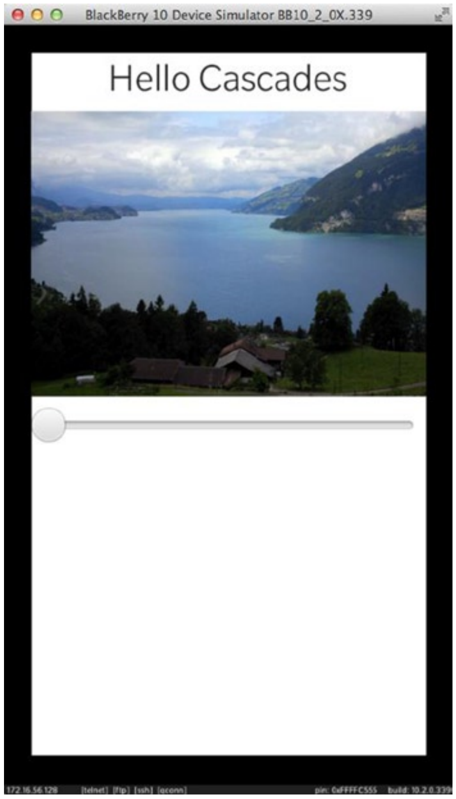


Figure 1-20. Hello Cascades on the simulator

6. Try moving the slider and notice how the scene changes from day to night.

To create a debug Launch configuration for the device, you basically need to repeat the same steps, with the following differences:

1. Set the active build configuration to Device-Debug.
2. Build the HelloCascades project.
3. Create a new launch configuration (see Figure 1-17 and Figure 1-18).
4. Give a name to your launch configuration (for example, HelloCascades Device-Debug).
5. Select the device target.
6. Press Debug.

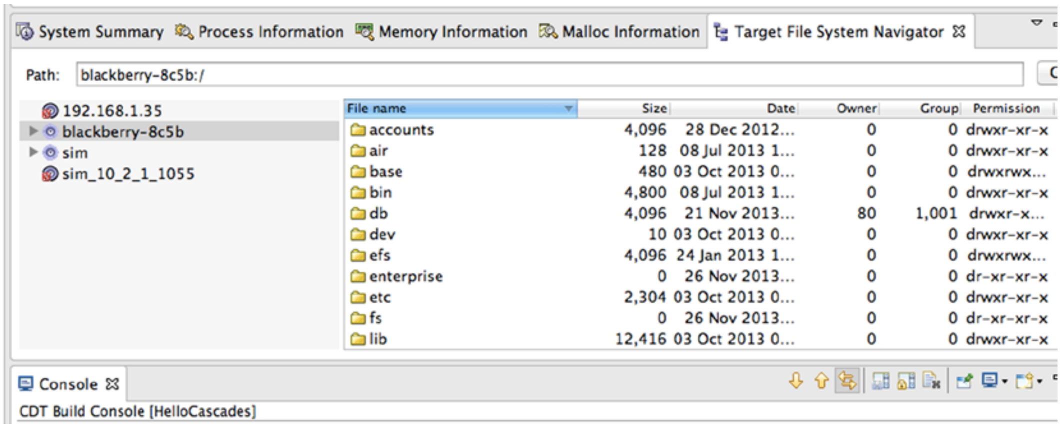
Once again, launch configurations can be completely ignored by using the BlackBerry Toolbar, but it is always a good idea to have a basic understanding of their functionality.

API Levels

An API level is a set of APIs and libraries that your application builds against. It also corresponds to a version of the BlackBerry 10 OS. API levels are backward compatible. (Higher API levels include APIs from the previous releases, although some APIs could be deprecated. In other words, this is identical to the way Java manages its APIs.) If for some reason you need to compile against a specific API level, you can change the setting in Momentics using Momentics ➤ Preferences ➤ BlackBerry ➤ API Level.

QNX System Information Perspective

Before finishing this chapter, I want mention the Momentics QNX System Information perspective, which can be used for navigating your device's or simulator's filesystem (you can open the perspective by selecting Windows ➤ Open Perspective ➤ QNX System Information; see Figure 1-21). As you develop Cascades applications, you will realize that the possibility to access your device will be extremely useful for retrieving logs from the target file system or for monitoring your application's memory and CPU usage.



Path: blackberry-8c5b:/

192.168.1.35
blackberry-8c5b
sim
sim_10_2_1_1055

File name	Size	Date	Owner	Group	Permission
accounts	4,096	28 Dec 2012...	0	0	drwxr-xr-x
air	128	08 Jul 2013 1...	0	0	drwxr-xr-x
base	480	03 Oct 2013 0...	0	0	drwxrwx...
bin	4,800	08 Jul 2013 1...	0	0	drwxr-xr-x
db	4,096	21 Nov 2013...	80	1,001	drwxr-x...
dev	10	03 Oct 2013 0...	0	0	drwxr-xr-x
efs	4,096	24 Jan 2013 1...	0	0	drwxrwx...
enterprise	0	26 Nov 2013...	0	0	dr-xr-xr-x
etc	2,304	03 Oct 2013 0...	0	0	drwxr-xr-x
fs	0	26 Nov 2013...	0	0	dr-xr-xr-x
lib	12,416	03 Oct 2013 0...	0	0	drwxr-xr-x

***Figure 1-21.** QNX System Information perspective*

Summary

This chapter gave you a bird's-eye view of the BlackBerry 10 platform and the Cascades programming model. I showed you how to declaratively design your UI using QML, which is much more efficient than using imperative C++ code. QML is therefore the preferred approach—not just because the Cascades framework takes care of a lot of the plumbing work for you, but also because you can rely on the Cascades Builder tools to visually design your UI. You can nevertheless still rely on C++, something that we will further discuss in Chapter 3, for the performance-critical aspects of your application. Signals and slots were introduced as a high-level mechanism used by Cascades for event handling and I explained how to use them in your own code for reacting to events generated by UI controls.

You discovered how the Momentics IDE was organized in Perspectives, giving a task-centric view of your work. The three most important ones are the QML Editing, C/C++ Editing, and Debug perspectives. You will be using them time and again when creating Cascades applications. We also went through the configuration of a BlackBerry device for development purposes, as well as the generation of the debug tokens required for application deployment on a device. Finally, you learned how to create, build, and launch configurations for your application in order to deploy it on a simulator or device.