Chapter 2

QML and JavaScript

QML and JavaScript are the cornerstones of Cascades declarative user interface design. Both technologies, while amazingly easy to master, pack an enormous amount of punch when it comes to creating user interfaces quickly and effortlessly. QML, being a declarative language, lets you describe your user interface much like HTML would describe a web page. JavaScript then adds programmatic logic in event handlers, slots in Qt/Cascades parlance, and essentially ties your UI together with some behavior.

I've deliberately kept C++ out of the mix because I want to exclusively concentrate on QML and JavaScript for the moment, but you will see in Chapter 3 that C++ also transparently integrates with QML. As a good rule of thumb, you should always rely on C++ whenever you need to access core platform services or do some heavy lifting, such as computationally intensive tasks.

> If you are a core JavaScript programmer and would like to quickly get a taste of Cascades programming, you can read this chapter, and then skip to Chapters 4 and 5 (this will provide you with the essential building blocks for creating Cascades applications using QML and JavaScript). At a later stage, you can return to Chapter 3 to understand what is happening behind the scenes in the C++ world.

QML, despite being a small language, is nevertheless extremely flexible, and by mastering the language's nuances, you will be able to build rich and enticing user interfaces. QML is also extensible: you can add to the core system your own types or "custom controls." You should consider this chapter as a review of the building blocks of QML, where you will learn how to assemble the language constructs and types to design your UI. Once you have mastered the basic elements of QML, you will be ready to apply them in full throttle in the following chapters. You will also have a firm grip on how Cascades uses the same language constructs for its own core controls (the topic of Chapters 4 and 5).

Syntax Basics

You have already seen in Chapter 1 an example of a QML document. I did not go into the details of explaining the QML syntax and I informally presented concepts such as properties, signals, and slots (or *signal handlers*, if you prefer). It is now time to dig a bit deeper and give you a description of the various QML syntactical elements.

QML Documents

A QML document is the basic building block for creating Cascades UIs. The QML syntax resembles JSON, except that you don't need to use quotes for defining attributes and that the QML language, combined with inline JavaScript, is much more expressive than JSON. Another big advantage of QML over other XML-based languages for designing UIs is that QML has been created from the ground up. The resulting language is very concise and expressive with advanced features, such as dynamic loading of components and transparent interoperability with C++ (you will see in Chapter 3 that you can very easily expose C++ objects to QML).

A QML document is a self-contained piece of QML source code that consists of the following:

- Import statements
- A root object declaration (the root object can also in turn declare children and JavaScript functions)

An example of a minimal `main.qml` document is given in Listing 2-1.

Listing 2-1. main.qml

```
import bb.cascades 1.0

Page {

}
```

As you saw in Chapter 1, the `main.qml` QML document is typically loaded during application start-up. The loading process is orchestrated behind the scenes by the QML declarative engine. When the engine encounters the `import bb.cascades 1.0` statement, it will also search through its import paths for the `bb.cascades` *namespace* and load the Cascades core controls and types registered with that namespace. By the time it reaches the `Page` object declaration, the QML engine already knows about the Page type definition, properties and signals, and is in measure to validate the Page element within the document.

Another interesting aspect of QML documents is that they provide an extension mechanism for defining new object types. In fact, a QML document implicitly defines a new type. For example, a document called `MyType.qml` will implicitly define the corresponding `MyType` QML type. The engine will also validate custom types declarations against their definition whenever you import them in other QML documents.

Import Statements

Import statements tell the declarative engine which libraries, resources, and component directories are used in a QML document. An import statement will then do any of the following:

- Load a versioned module containing QML registered types. This is how you import the Cascades core controls module in the global namespace (this is also how C++ types are exposed to QML through the `qmlRegisterType` method, as we will see in Chapter 3).
- Specify a directory relative to your application's assets directory, which contains type definitions in QML documents. As a result, all the QML object types defined in the directory will be imported in the global namespace.
- Load a JavaScript file containing functions that you want to use in your QML document.

When using an import statement, you can further use a qualifier as a local namespace identifier. This is mostly used when importing a JavaScript file or when you want to make sure that there will be no clashes with types declared in the global namespace. Listing 2-2 shows a few examples:

Listing 2-2. imports

```
import bb.cascades 1.2

import "mycontrols"
import "mycontrols/core" as MyCoreControls
import "parser.js" as Parser
```

The first line imports the versioned `bb.cascades` library (or module) in the global QML document namespace. The second line imports all the QML types defined in the `mycontrols` directory in the global QML namespace. The third example imports the QML types defined in the `mycontrols/core` directory and binds them to the local `MyCoreControls` namespace. A type `SomeType` will then be accessible using `MyCoreControls.SomeType`. This is essentially a way of avoiding clashes when importing controls with the same name from different modules (for example, if you have defined your own Label control in `mycontrols/core`, then it will not clash with the Cascades control with the same name and yours will be accessible using `MyCoreControls.Label`).

Object Declarations

In `main.qml`, a block of QML code defines a scene graph of objects to be created by the runtime engine. An object is declared in QML using the name of its object type followed by a set of curly braces. The object's attributes are then declared in the body. An object's attribute can in turn be another object declaration. In this case, you simply need to reapply the same rules for declaring that attribute. Listing 2-3 extends the example given in Listing 2-1 to show you how this works in practice.

Listing 2-3. main.qml

```
import bb.cascades 1.2

Page {
    id: mainscreen
    content: Container {
        id: maincontainer
        controls: [
            Button {
                id: first
                text: "Click me!"
            },
            Button {
                id: second
                text: "Not me!"
            }
        ]
    }
}
```

> The `import bb.cascades 1.2` statement in Listing 2-3 tells the QML engine to import version 1.2 of the Cascades library, which is provided in BlackBerry API level 10.2. If you are targeting a different API level, the import statement should reflect the corresponding Cascades library version (for example, `import bb.cascades 1.0` provided in BlackBerry API level 10.0).

The Page control represents a device's screen. Its content property, the root control, is usually a cascades Container core control. A Container can in turn include child controls as well as other Containers by setting its `controls` *property* (note that the property values are specified in brackets ([ ]), indicating that this is a QML list). As you will see later in this chapter, QML objects also have a predefined id property, which is useful when you want to reference them from JavaScript code. The page declaration in Listing 2-3 is a bit verbose and you can actually make it shorter by avoiding explicitly declaring *default properties*. A property can be marked as "default" in the QML object type *definition*, and whenever you declare an object without assigning it to a property, the QML engine will try to assign it to the parent object's default property (the default property for the `Page` control is `content` and the default one for `Container` is `controls`). Listing 2-4 gives an updated version of `main.qml` using default properties.

Listing 2-4. main.qml with Default Properties

```
import bb.cascades 1.2

Page {
    id: mainscreen
    Container {
        id: maincontainer
        Button {
                id: first
                text: "Click me!"
        }
```

```
        Button {
                id: second
                text: "Not me!"
        }
    }
}
```

QML Basic Types

This section reviews some of the most important QML basic types that you will often use when writing Cascades applications.

- `string`, `int`, `bool`, `real`, `double`: The "standard" types supported by most programming languages. A `real` is a number with a decimal point. A `double` is a number stored in double precision.
- *list*: A list type contains a list of objects. In JavaScript, you can use the standard array notation to access the list elements. For example, `myList[0]`. You can also use the length property for iteration: `for (int i=0; i < myList.length; i++ ) {...}`.
- *enumeration*: An enumeration type consists of a set of named values. For example, the `LayoutOrientation` enumeration, which dictates how controls are displayed on the screen, can take values such as `LayoutOrientation.TopToBottom` or `LayoutOrientation.LeftToRight`.
- *variant*: A generic type that can contain any of the other basic types.

Creating a Custom Control

The best way to learn QML is by designing a custom type or control. This will give you the opportunity to see how the different QML syntactical elements fit together and will also give you some insight on how they are used by the Cascades framework.

If you've worked in a large corporation, chances are that you have already relied on an intranet for locating a person in your organization. This information is usually stored in LDAP directories and accessed by using a client application over an intranet. When you look up a person's entry, you will usually be presented with his surname, first name, job title, employee number, and so forth. You will also be presented with the person's picture so that you can easily recognize him when you attend one of those boring corporate meetings. Now let us imagine that your organization has decided to maintain this information using BlackBerry devices. You have been tasked to design a reusable custom control for displaying and updating a person's entry. Let us start by defining a new QML type called `PersonEntry`.

1. Create a new standard empty Cascades project and call it `CorpDir`.
2. In the `assets` folder of your project, where `main.qml` is located, create a new QML file called `PersonEntry.qml` using the Container template (right-click the `assets` folder, and then select New ➤ QML File).
3. Set the Container control's id to root (see Listing 2-5).

Listing 2-5. PersonEntry.qml

```
import bb.cascades 1.0

Container{
    id: root

}
```

4. `PersonEntry` is now a new, albeit minimal, custom type recognized by the QML declarative engine. Also, because `PersonEntry`'s root control is a Container, you can add it to a QML page. Go ahead and modify `main.qml` as shown in Listing 2-6.

Listing 2-6. main.qml

```
import bb.cascades 1.0

Page {
   PersonEntry {

    }
}
```

Attributes

Let's go over the attributes used in the example.

The id Attribute

As mentioned previously, object declarations can specify an `id` attribute that must start with a lowercase letter or an underscore. You will usually assign a value to the `id` attribute whenever you want to uniquely reference that object instance in your QML document.

Property Attributes

Let us now flesh out our `PersonEntry` type by adding some *properties* to it. We want to be able to add extra information such as a person's surname, first name, login, e-mail, and so forth. We will also eventually have to implement business rules such as "a person's login is the first letter of his first name concatenated to his surname with all letters in lowercase" and "a person's e-mail is his login followed by the at symbol and the company's domain name."

In QML, this kind of information is provided by object properties.

A *property* is an attribute that can be assigned a static value or bound to a dynamic expression provided by some JavaScript code. Properties are used to expose to the "outside world" an object's state by hiding its implementation at the same time. The syntax for defining properties is given by

```
property <propertyType> <propertyName>
```

Listing 2-7 adds the surname, first name, login, and e-mail properties to PersonEntry.qml.

Listing 2-7. PersonEntry.qml

```
import bb.cascades 1.0

Container{
    id: root
    property int employeeNumber
    property string surname
    property string firstname
    property string login
    property string email

}
```

You can in turn set the properties in `main.qml` as shown in Listing 2-8.

Listing 2-8. main.qml

```
import bb.cascades 1.0

Page {
    PersonEntry {
        employeeNumber: 100
        surname: "Smith"
        firstname: "John"
        login: "jsmith"
        email:"jsmith@mycompany.com"
    }
}
```

The `PersonEntry` control is visually not very interesting at the moment. The most glaring problem is that it's missing a screen representation. If you try to build the project and run it on the simulator, you will just get a blank screen (you will notice the same thing in the "Cascades builder" design view if you try to display `main.qml`). What we need to do is display the properties on the screen after they have been set. In order to achieve this, let's use Cascades Labels (a Label is a core control with a displayable text property on the screen). Listing 2-9 gives an updated version of `PersonEnty.qml` using Labels for displaying the object's properties.

Listing 2-9. PersonEntry.qml

```
import bb.cascades 1.0

Container{
    id: root
    property int employeeNumber
    property string surname
    property string firstname
    property string login
    property string email

    Label{
        text: "MyCompany Employee Details"
        textStyle.base: SystemDefaults.TextStyles.TitleText
        horizontalAlignment: HorizontalAlignment.Center
    }
```

```
    Label{
        text: "Employee number: " + employeeNumber;
    }

    Label{
        text: "Last name: "+surname;
    }

    Label{
        text: "First name:"+ firstname;
    }

    Label{
       text: "Login: "+ login;
    }
    Label{
        text: "Email: "+ email;
    }

}
```

You can now build the `CorpDir` project and run on it on the simulator (see Figure 2-1).

***Figure 2-1.** Employee view*

So far so good, but we still need to be able to set the person's job title. This kind of information usually comes from a list of predefined values such as Software Engineer, Manager, Director, Consultant, Technician, and so forth. In order to achieve this, we can use a Cascades DropDown control. The control will be selectable so that if a person's entry needs to be updated, a new job title can be selected from the list. See Listing 2-10 for the updated `PersonEntry.qml` control.

Listing 2-10. PersonEntry.qml

```
import bb.cascades 1.0

Container {
    id: root
    property int employeeNumber
    property string surname
    property string firstname
    property string login
    property string email
    property string jobTitle

    Label{
        text: "Employee Details"
        textStyle.base: SystemDefaults.TextStyles.TitleText
        horizontalAlignment: HorizontalAlignment.Center
    }

    Label {
        text: "Employee number: " + employeeNumber;
    }

    Label {
        text: "Last name: " + surname;
    }

    Label {
        text: "First name:" + firstname;
    }

    Label {
        text: "Login: " + login;
    }
    Label {
        text: "Email: " + email;
    }
    DropDown {
        id: jobs
        title: "Job Title"
        enabled: true
        Option{
            text: "Unknown"
        }
```

```
        Option {
            text: "Software Engineer"
        }
        Option {
            text: "Manager"
        }

        Option {
            text: "Director"
        }

        Option {
            text: "Technician"
        }

    }

}
```

Listing 2-11 shows `main.qml` with the job property set.

***Listing 2-11.** main.qml*

```
import bb.cascades 1.0

Page {
    PersonEntry {
        employeeNumber: 100
        surname: "Smith"
        firstname: "John"
        login: "jsmith"
        email:"jsmith@mycompany.com"
        jobTitle: "Software Engineer"
    }
}
```

At this point, we are still facing a couple of issues. For one thing, we need to be able to synch the `jobTitle` property with the corresponding option value in the DropDown control. Also, instead of setting the e-mail and login properties, they should be generated using the business rules described at the start of this section. Whenever you need to add this kind of programmatic logic, you will have to rely on JavaScript.

JavaScript

JavaScript is not an object attribute per se, but is still tightly integrated and can be used in the following scenarios:

- A JavaScript expression can be bound to QML object properties. The expression will be reevaluated every time the property is accessed in order to ensure that its value stays up-to-date. Typically, the one-liners you have seen until now for setting a Label's property are JavaScript expressions (for example, `"Login" + login;`). The expressions can be as complex as you wish as long as

their result is a value whose type can be assigned to the corresponding property. You can even include multiple expressions between open and close braces.

- Signal handlers can contain JavaScript code that is automatically evaluated every time the corresponding QML object emits the corresponding signal.
- You can define custom JavaScript methods within a QML object (this can be considered as an object attribute).
- You can import JavaScript files as *modules* that you can use in your QML document.
- And finally, you can wire a signal directly to a JavaScript function.

You have already encountered the first two methods of using JavaScript, and by time you finish this chapter, you will have seen all the different ways of incorporating JavaScript in your QML documents.

JavaScript Host Environment

The QML engine includes a JavaScript host environment, giving you the possibility of building extremely complex applications using JavaScript/QML only. There are some restrictions, however; for example, the environment does not provide the DOM API commonly available in browsers. If you think about it, this makes complete sense since a QML application is certainly not an HTML browser app and the DOM would be irrelevant. Also, the environment is quite different from server-side technologies such as Node.js. The runtime does, however, implement the ECMAScript language specification, so this effectively means that you have a complete JavaScript programming environment at your disposal. The host environment also provides a set of global objects and functions that you can use in your QML documents:

- The `Qt` object, which provides string utility functions for localization, date formatting functions, and object factories for dynamically instantiating Qt types in QML.
- The `qsTr()` family of functions for providing translations in QML.
- The `console` object for generating logs and debug messages from QML (using `console.log()` and `console.debug()`).
- And finally, the `XMLHttpRequest` object. This basically opens the door to asynchronous HTTP requests directly from QML!

Let us now return to our `PersonEntry` type and spice it up with some JavaScript behavior (see Listing 2-12).

Listing 2-12. PersonEntry.qml

```
import bb.cascades 1.0
Container {
    id: root
    property int    employeeNumber
    property string surname
    property string firstname
```

```
property string jobTitle

function getLogin(){
    return root.firstname.charAt(0).toLowerCase() + root.surname.toLowerCase();
}

function getEmail(){
    return root.firstname.toLowerCase() +"."+root.surname.toLowerCase()+"@mycompany.com";
}

onCreationCompleted: {
    switch (jobTitle) {
        case "Software Engineer":
            jobs.selectedIndex = 1;
            break;
        case "Manager":
            jobs.selectedIndex = 2;
            break;
        case "Director":
            jobs.selectedIndex = 3;
            break;
        case "Technician":
            jobs.selectedIndex = 4;
            break;
        default:
            jobs.selectedIndex = 0;
           break;
    }
}

Label{
    text: "Employee Details"
    textStyle.base: SystemDefaults.TextStyles.TitleText
    horizontalAlignment: HorizontalAlignment.Center
}

Label {
    text: "Employee number: " + employeeNumber;
}

Label {
    text: "Last name: " + surname;
}

Label {
    text: "First name:" + firstname;
}

Label {
    text: "Login: " + root.getLogin();
}
```

```
    Label {
        text: "Email: " + root.getEmail();
    }

    DropDown {
        id: jobs
        title: "Job Title"
        enabled: true

        onSelectedIndexChanged: {
            console.debug("SelectedIndex was changed to " + selectedIndex);
            console.debug("Selected option is: " + selectedOption.text);
            root.jobTitle = selectedOption.text;
        }

        Option{
            text: "Unknown"
        }

        Option {
            text: "Software Engineer"
        }

        Option {
            text: "Manager"
        }

        Option {
            text: "Director"
        }

        Option {
            text: "Technician"
        }

    }

}
```

Listing 2-13 is the updated version of `main.qml`.

Listing 2-13. main.qml

```
import bb.cascades 1.0

Page {
    PersonEntry {
        employeeNumber: 100
        surname: "Smith"
        firstname: "John"
        jobTitle: "Jack of All Trades"
    }
}
```

You will notice that `login` and `email` are no longer settable properties. Instead, the `getLogin()` and `getEmail()` JavaScript functions are used in order to update the corresponding labels using the business rules for generating logins and e-mails respectively. Another interesting point is that in order to synchronize the `jobFunction` property with the DropDown control's selected index, the `onCreationCompleted:` signal handler is used (the body of the handler is simply a switch statement that sets the selected index). The QML engine automatically calls this handler after a QML object has been successfully constructed. This is the ideal place to set up additional validation or initialization logic (in the example given in Listing 2-13, "Jack of All Trades" is not a valid job title and the `selectedIndex` will be set to 0, which corresponds to the "Unknown" job title).

Signal Attributes

In Chapter 1, you declared signals in C++ using the `signals:` annotation. Declaring your own signals in QML is just as simple and is given by the following syntax:

```
signal <signalName>[([<type> <parameter name>[, ...]])]
```

If your signal does not take any parameters, you can safely ignore the "`()`" brackets in the declaration.

Here are two examples:

- `signal clicked`
- `signal salaryChanged(double newSalary)`

There are also a couple of things that the QML engine provides you "for free:"

- The QML engine generates a slot for every signal emitted by your controls. For example, the `onSalaryChanged` slot will be generated for the `salarayChanged` signal (you will see this in action in Listing 2-15).
- Property change signals. The QML engine automatically generates these signals for your custom control's properties. They are emitted whenever a control's property value is updated.
- Property change signal handlers. For a given property `<Property>`, they take the form `on<Property>Change`. This is where you can define your own business logic when the property change signals are emitted.

Let's add the `salaryChanged` signal to the `PersonEntry` control and the corresponding handler in `main.qml`. The signal will be emitted with an updated salary whenever a person's job title changes. The first step is to define the signal in the root QML object. You can then emit the signal using `root.salaryChanged()` from the DropDown control's `onSelectedIndexChanged` handler. The final version of the `PersonEntry` custom control also includes a new property for setting the person's picture. (Note that I am using a *property alias* in this case. A property alias is a reference to an existing property. In other words, the `picture` property is a reference to the `employeeImage.imageSource` property, and by setting the `picture` property, you are actually updating the referenced property.)

Listing 2-14. PersonEntry.qml Final

```
import bb.cascades 1.0
Container {
    id: root
    property int employeeNumber
    property string surname
    property string firstname
    property string jobTitle
    property alias  picture: employeeImage.imageSource

    signal salaryChanged(double newSalary)

    function getLogin(){
        return root.firstname.charAt(0).toLowerCase() + root.surname.toLowerCase();
    }

    function getEmail(){
        return root.firstname.toLowerCase() +"."+root.surname.toLowerCase()+"@mycompany.com";
    }

    onCreationCompleted: {
        switch (jobTitle) {
            case "Software Engineer":
                jobs.selectedIndex = 1;
                break;
            case "Manager":
                jobs.selectedIndex = 2;
                break;
            case "Director":
                jobs.selectedIndex = 3;
                break;
            case "Technician":
                jobs.selectedIndex = 4;
                break;
            default:
                jobs.selectedIndex = 0;
        }
    }

    ImageView {
        id: employeeImage
        horizontalAlignment: HorizontalAlignment.Center
    }

    Label{
        text: "Employee Details"
        textStyle.base: SystemDefaults.TextStyles.TitleText
        horizontalAlignment: HorizontalAlignment.Center
    }

    Label {
        text: "Employee number: " + employeeNumber;
```

```
}

Label {
    text: "Last name: " + surname;
}

Label {
    text: "First name:" + firstname;
}

Label {
    text: "Login: " + root.getLogin();
}
Label {
    text: "Email: " + root.getEmail();
}
DropDown {
    id: jobs
    title: "Job Title"
    enabled: true

    onSelectedIndexChanged: {
        console.debug("SelectedIndex was changed to " + selectedIndex);
        console.debug("Selected option is: "+selectedOption.text);
        root.jobTitle = selectedOption.text;
        switch (selectedOption.text){
            case "Software Engineer":
                root.salaryChanged(90000);
                break;
            case "Manager":
                root.salaryChanged(100000);
                break;
            case "Director":
                root.salaryChanged(150000);
                break;
            case "Technician":
                // yes technicians should be more rewared than Managers
                // as they are more useful.
                root.salaryChanged(160000);
                break;
            default:
                root.salaryChanged(0.0);
        }
    }

    Option{
        text: "Unknown"
    }

    Option {
        text: "Software Engineer"
    }
```

```
        Option {
            text: "Manager"
        }

        Option {
            text: "Director"
        }

        Option {
            text: "Technician"
        }

    }

}
```

And Listing 2-15 gives the final version of `main.qml`. You will notice that now the root control is no longer a `PersonEntry` object but a Container. The reason for this is because we have also added a Label that will display a person's updated salary whenever the `salaryChanged` signal is emitted.

Listing 2-15. main.qml Final

```
import bb.cascades 1.0

Page {
    Container{
        PersonEntry {
            employeeNumber: 100
            surname: "Smith"
            firstname: "John"
            jobTitle: "Jack of All Trades"
            picture: "asset:///johnsmith.png"

            onSalaryChanged: {
                salaryLabel.text = "Salary: "+newSalary;
            }
        }
        Label{
            id: salaryLabel

        } // Label
    } // Container
} // Page
```

You can now finally build the `CorpDir` application and run it on the simulator (see Figure 2-2).

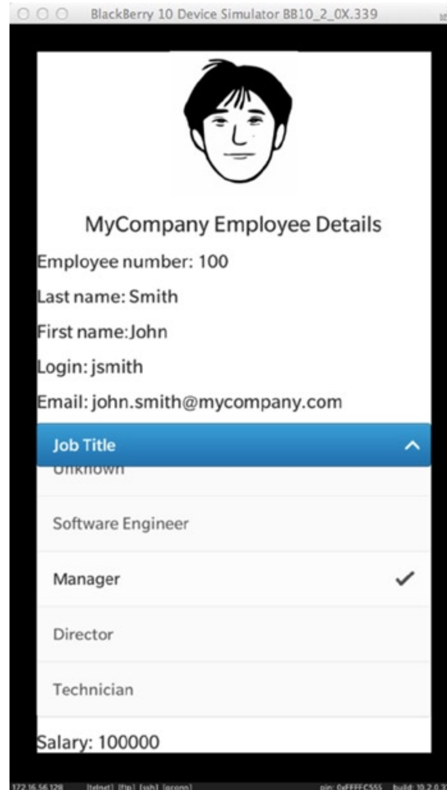

Figure 2-2. Employee view

XMLHttpRequest Example

In this section, I want to show you how easily you can use the `XMLHttpRequest` object from QML. The sample code provided here is a quick and dirty REST client for the Weather Underground weather forecast service (`www.wunderground.com/weather/api/d/docs`). To call the REST service, you will need to register and obtain a free development key. Listing 2-16 shows you how to use the service to get a weather forecast for a given city. The application is quite basic at the moment and simply “dumps” the result of the query in a `TextArea` (see Figure 2-3; you will see in Chapter 7 how to enhance the app by building a full-fledged weather service client). The most important point to keep in mind is that the call to the weather service is completely *asynchronous* and will not block the UI thread.

Listing 2-16. main.qml

```
import bb.cascades 1.0

Page {
    id: root
    function getWeather(apikey, city, state) {
        var getString = "http://api.wunderground.com/api/"+apikey+"/conditions/q/";
        if("".valueOf() != state.valueOf()){
            getString = getString+state;
        }
```

```
        getString = getString + "/"+city+".json";
        var request = new XMLHttpRequest();
        request.onreadystatechange = function() {
            // Need to wait for the DONE state or you'll get errors
            if (request.readyState === XMLHttpRequest.DONE) {
                if (request.status === 200) {
                    result.text = request.responseText
                } else {
                    // This is very handy for finding out why your web service won't talk to you
                    console.debug("Status: " + request.status + ", Status Text: " + request.
statusText);
                }
            }
        }
        request.open("GET", getString, true); // only async supported
        request.send()
    }
    Container {
        Container {
            layout: StackLayout {
                orientation: LayoutOrientation.LeftToRight
            }
            TextField {
                id: locationField
                layoutProperties: StackLayoutProperties {
                    spaceQuota: 2
                }
            }
            Button {
                text: "Get City!"
                layoutProperties: StackLayoutProperties {
                    spaceQuota: 1
                }
                onClicked: {
                    var values = locationField.text.split(",")
                    if(values.length > 1){
                        root.getWeather("75cfd4c741088bfd", values[0], values[1]);
                    }
                    else
                    root.getWeather("75cfd4c741088bfd", values[0],"");
                }
                verticalAlignment: VerticalAlignment.Center
            }

        }
        ScrollView {
            TextArea {
                id: result

            }
        }
    }
}
```

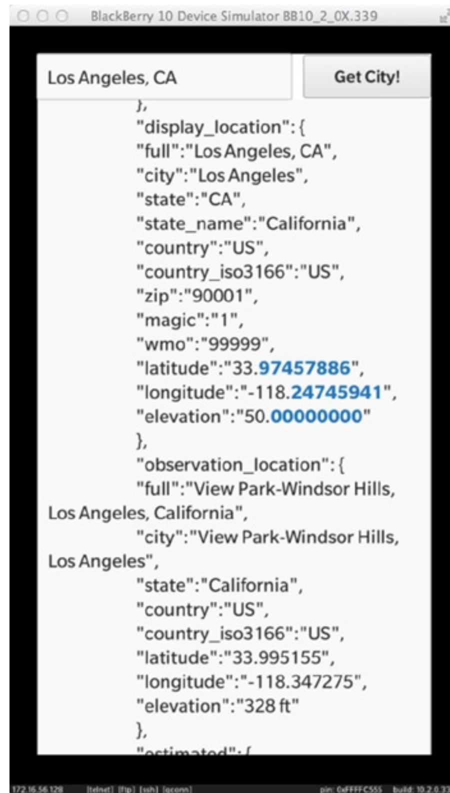

***Figure 2-3.** Weather service*

When the user touches the Get City! button, its `onClicked` slot calls the `getWeather()` JavaScript function defined at the Page level. The function in turn uses the standard `XMLHttpRequest` object to asynchronously call the weather service. An anonymous callback function is also provided in order to handle the `HTTP` response and update the `TextArea` (note that this the standard `AJAX` way of handling requests and responses).

SCalc, the Small Calculator

Before wrapping up this chapter, I want to illustrate how QML and JavaScript can be used for developing a slightly more complex application than the ones shown up to this point. This will also give me the opportunity to explain your application's project structure, something that I skimmed over in Chapter 1 (if you want to give the application a try right away, you can download it from BlackBerry World).

> To import the SCalc project in Momentics, you can clone the https://github.com/aludin/BB10Apress repository.

SCalc is a simple calculator app written entirely in JavaScript and QML (see Figure 2-4). The application's UI is built in QML and the application logic is handled in JavaScript using Matthew Crumley's JavaScript expression engine (https://github.com/silentmatt/js-expression-eval). The engine is packaged as a single JavaScript file that I have dropped in the project's `assets` folder (`parser.js`, located at the same level as `main.qml`).

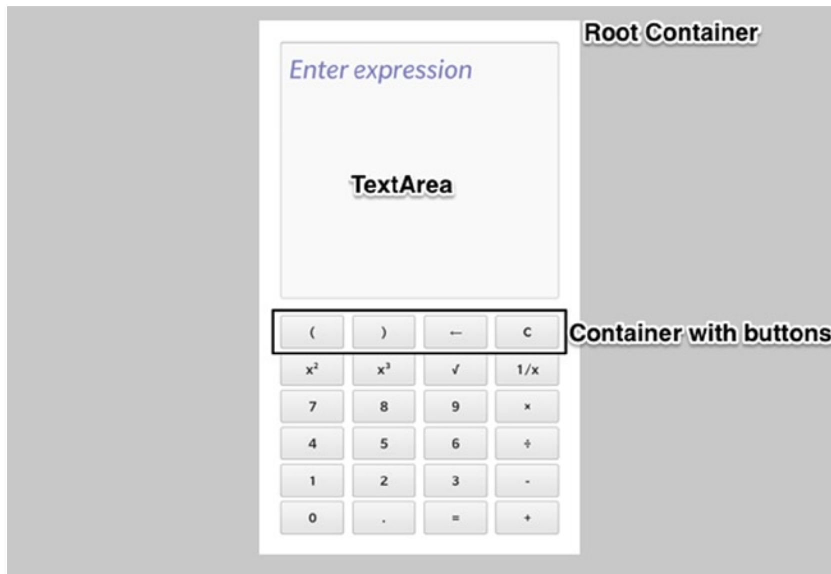

***Figure 2-4.** SCalc*

As illustrated in Figure 2-4, a root Container contains a `TextArea` and six child containers, which in turn hold Button controls. Finally, the parent of the root Container is a Page control that represents the UI screen. Another way of understanding the control hierarchy is by looking at the outline view in the QML perspective in Momentics (see Figure 2-5).

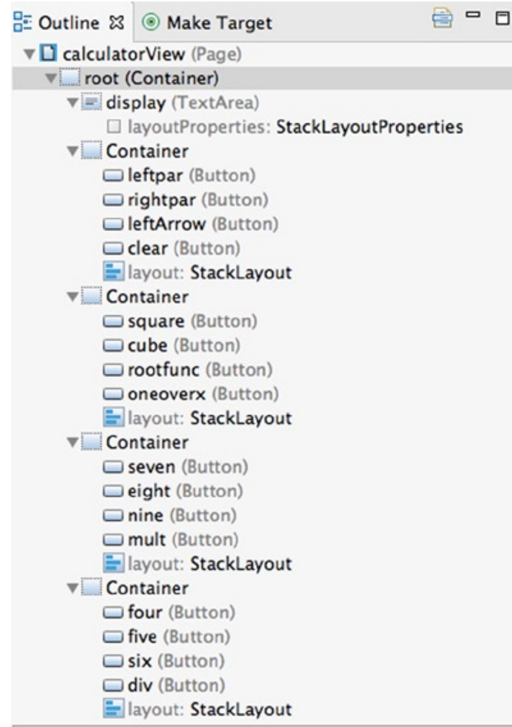

***Figure 2-5.** Outline view (four child Containers below root shown)*

You will notice that the containers have a layout property. A layout is an object that controls the way UI elements are displayed on the screen. In our case, we are using a StackLayout, which stacks controls horizontally or vertically. The root container does not define a layout and therefore Cascades will assign it a default StackLayout that stacks controls vertically. (In the child containers, the layout orientation has been set to horizontal, thus displaying the buttons in a row. I will tell you more about layout objects in Chapter 4.) Listing 2-17 is an outline of `main.qml`.

***Listing 2-17.** main.qml*

```
import bb.cascades 1.2
import "parser.js" as JSParser
Page {
    id: calculatorView
    // root containter goes here
}
```

The second import statement imports the JavaScript expression engine as a module and assigns it to the `JSParser` identifier (because the file is located in the same folder as `main.qml`, you don't need to provide a path to the file). Now that the library has been imported, you will be able to use it in your QML document (the expression engine provides a `Parser` object that you can call using `JSParser.Parser.evaluate(expression, "")`).

As mentioned previously, the root container contains a `TextArea` and six child Containers (see Listing 2-18).

Listing 2-18. root Container

```
 Container {
        id: root
        // padding properties omitted.
         layout: StackLayout {
            orientation: LayoutOrientation.TopToBottom
        }
         // background properties and attached object omitted.
        TextArea {
            bottomMargin: 40
            id: display
            hintText: "Enter expression"
            textStyle {
                base: SystemDefaults.TextStyles.BigText
                color: Color.DarkBlue
            }
            layoutProperties: StackLayoutProperties {
                spaceQuota: 2
            }
        }
        Container{ // 1st row of Buttons, see Figure 2-4
               Button{id: lefpar ...}
               Button{id: rightpar ...}
               Button{id: leftArrow ...}
               Button{id: clear ...}
}
       ... // 5 more Containers
}
```

The application logic is implemented by handling the clicked signal emitted by the Buttons and by updating the TextView with the current expression. For example, Listing 2-19 shows the implementation for the clicked signal when Button “7” is touched by the user.

Listing 2-19. Button “7”

```
Button {
    id: seven
    text: "7"
    onClicked: {
        display.text = display.text + 7;
    }
}
```

Finally, the JavaScript expression library’s `Parser.evaluate()` method is called when the user touches the “=” button (the TextView is also updated with the result of the evaluation).

Listing 2-20. Button “=”

```
Button {
    id: equal
    text: "="
    onClicked: {
```

```
        display.text = JSParser.Parser.evaluate(display.text, "");
    }
}
```

The current input handling logic is quite crude and you can easily enter invalid expressions. As an exercise, you can try to make it more robust.

Project Structure

Figure 2-6 illustrates SCalc's project structure in the Momentics Project Explorer view.

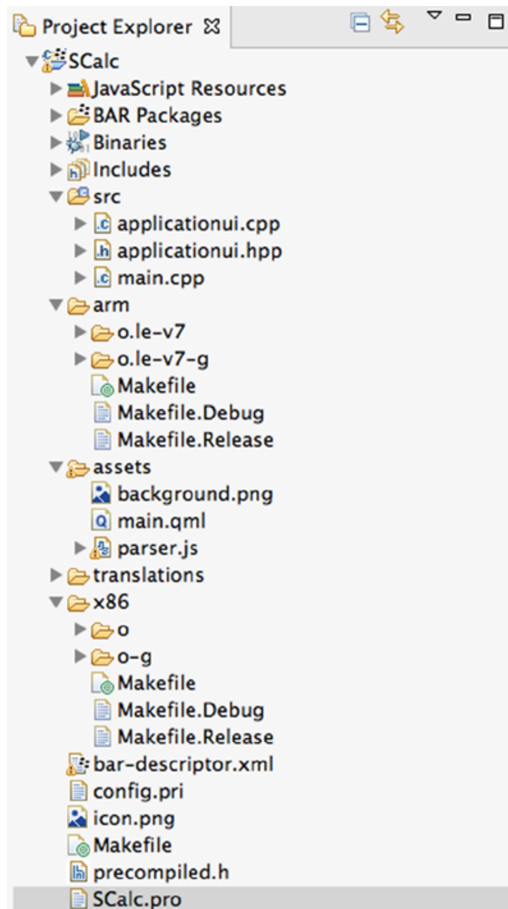

***Figure 2-6.** Project Explorer*

You will find the same structure in all of your applications and you should therefore take some time to get familiar with it. Here is a quick overview of the most important project elements:

- `src`: You will find the C++ source code of your project in this folder.
- `assets`: This folder contains your QML documents. You can also create subfolders and include additional assets such as images and sound files. You will generally load at runtime the assets located in this folder.
- `x86`, `arm`: Folders used for the build results of your application for the simulator and device respectively. The folders include subfolders, depending on the build type (debug or release). For example, a debug build for the simulator will be located under `\x86\o-g` (and the corresponding Device folder is `\arm\o.le-v7-g`).
- `SCalc.pro`: This is your project file and includes project settings. You can add dependencies such as libraries in this file (you will see how this works in Chapter 3).
- `bar-descriptor.xml`: This file defines important configuration settings for your application. In particular, it also defines your application's permissions. You will have to update this file for certain projects in following chapters. The easiest way to proceed is to work with the General and Permissions tabs when the file is open in Momentics (see Figure 2-7).

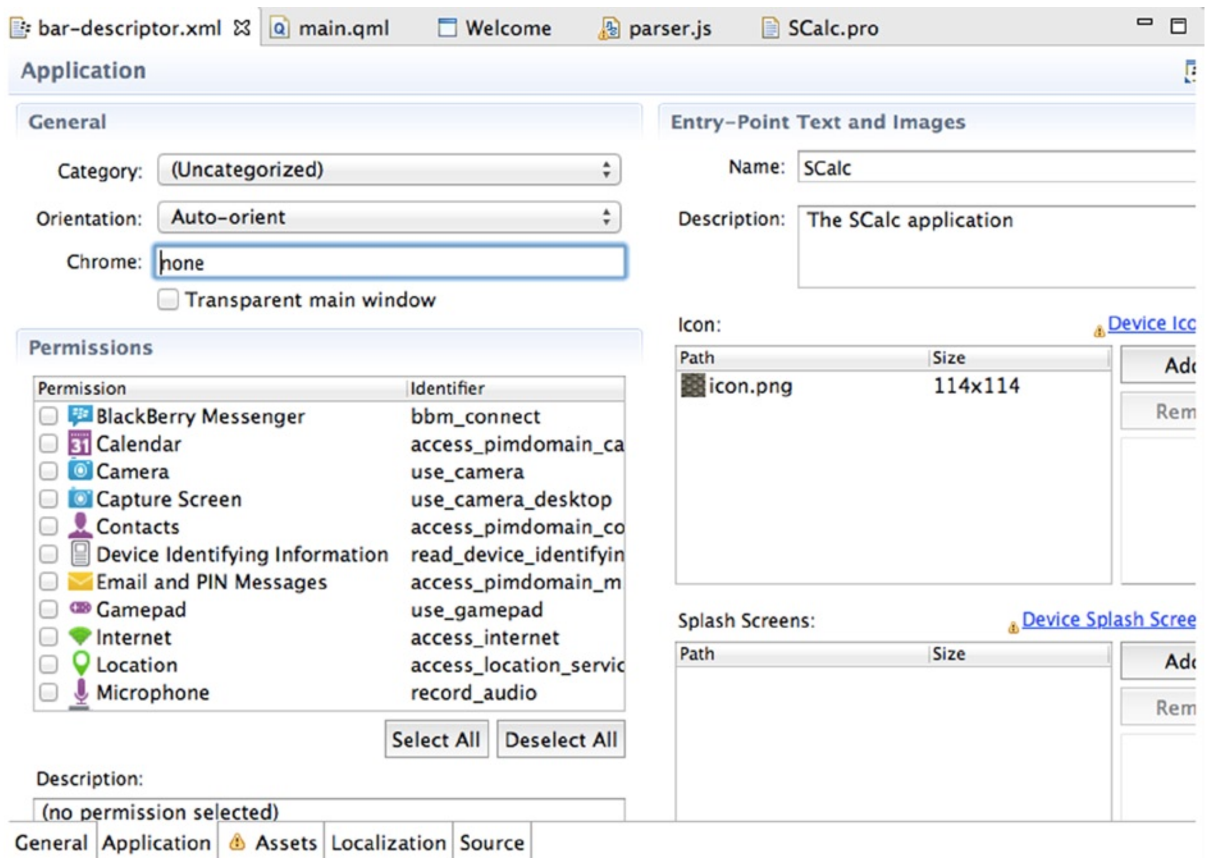

Figure 2-7. bar-descriptor.xml view

- `icon.png`: Your application's icon.

Summary

This chapter dissected the core elements of the QML language. You discovered how the different elements of QML fit together by designing your own custom control. You also saw that QML, despite its simplicity, is an extremely powerful programming environment. Most importantly, this chapter gave you some insight on how Cascades uses those same QML constructs, and hopefully unveiled some of the magic involved in Cascades programming.

JavaScript is the glue giving you the tools for adding some programmatic logic to your controls. The environment provided by QML runtime is ECMAScript compliant. This means that at this point you can build full-fledged Cascades applications using QML and JavaScript.