symbian Developer Network

Multi-Language Programming

Part IV - RESTful Widgets

Bernd Wiegmann

Published by the Symbian Developer Network

Version: 1.0 – December 2008

1 INTRODUCTION 2
2 EXTENDING WIDGETS WITH SYMBIAN C++ OR PYTHON 2
3 ARCHITECTURE FOR MULTI-RUNTIME PROJECTS WITH WIDGETS 3
4 WEB SERVICES 4
5 RESTFUL WIDGET DESIGN CONSIDERATIONS 6
6 AN EXAMPLE OF A RESTFUL WIDGET 7
7 INTEGRATION 11
8 SECURITY 12
9 CONCLUSION 12
10 REFERENCES 13
11 AUTHOR PROFILE 14

1 Introduction

This is the fourth and final paper in a series examining the possibilities of building Symbian OS applications by mixing multiple languages and runtime environments. The first paper [R1] presented an overview of the runtimes that are available and the combinations of them which make sense, while the second focused on extending Flash Lite with Java ME or Symbian C++ [R2], and the third discussed combining Python with Flash Lite and Symbian C++ [R3].

In this paper we focus on Widgets, applications that use the Web Runtime (WRT) [R4]. The Web Runtime is new to mobile but provides an environment similar to that of Apple OS X Dashboard Widgets or Windows Vista Gadgets. Widgets essentially use the web technologies of XHTML, CSS and JavaScript to provide a familiar environment for web designers and programmers. Widgets can be used to their full potential as thin clients for Web Services on the Internet.

As with Flash Lite and Java ME, Widgets can benefit from multi-language programming because they have limited access to resources on the phone and in most cases require an active internet connection. Combining Widgets with a local Symbian C++ or Python server on the phone results in a powerful combination that negates the limitations of Widgets.

The S60 5th Edition SDK introduced new APIs for JavaScript [R5] which can be used by the Web Runtime to access local phone features such as sensors, GPS and the phonebook. Despite these new features, combining Widgets with other runtimes is still useful for enabling functionality like a background server that is waiting for an incoming SMS or collecting location data.

In contrast to standalone Widgets, the combination approach can be used to create richer clients to Web Services along the lines of other emerging technologies like Google Gears [R6].

2 Extending Widgets with Symbian C++ or Python

Using multiple runtimes in a project increases its complexity and requires broader skills than just using a single technology. There is no universal answer to the question of whether you should use a combination of runtimes or not, however the first part of this paper discusses some of the motivations to do so.

2.1 Web Technologies

The programming language used for Widgets is JavaScript. It is quite a powerful language to support interactive web pages, which is basically what a Widget is. On the other hand JavaScript was not designed for large programs and has no strong support for modularization in the way that Symbian C++ or Python do.

JavaScript has no built-in limitations regarding the size of an application, but for any application which requires more complex functionality it makes sense to think about a multi-runtime combination.

2.2 Internet Connection

Using an Internet connection on a mobile device has some drawbacks, the first of which us that the connection itself is not very stable. As a Widget relies on a stable connection, this may lead to poor user experience. Furthermore, even when the connection is good, it requires constant power to use the WLAN or 3G, which results in a shorter battery life.

Using a local cache for content retrieved over the Internet results in a better user experience with increased responsiveness and longer battery life. For a weather or stock quote Widget, caching

offers little advantage because these Widgets rely on frequently updated information. However, a Widget to look up train timetables uses mainly static information which may only change about twice a year and, in such cases, caching is a viable solution.

2.3 Access to Local Phone Functionality

The first version of the Web Runtime for S60 3rd Edition FP2 and selected FP1 devices offered no access to location functionality such as sensors and GPS. The updated version for the S60 5th Edition offers additional APIs for this purpose and it can be expected that the successive releases will offer even more.

This extended API support allows developers to create feature-rich Widgets, but limitations remain for use cases such as running a background server. To achieve this kind of functionality with a Widget, you must combine it with Symbian C++ or Python has to be used.

2.4 Python or Symbian C++?

Symbian C++, the native language of Symbian OS, is the most powerful available, but for several reasons it also has the steepest learning curve. PyS60 is a port of the widely available language Python and it requires much less Symbian-specific knowledge to create applications for Symbian phones. [R1] and [R3] give some information about Python in a multi-runtime context and information about Python on Symbian can also be found at [R7], [R8] and [R9].

From the perspective of extending Widgets, neither of these two languages offers significant advantages over the other and the final choice depends largely on your project's requirements. Symbian C++ provides better coverage of system services and allows easier installation; however, it is usually much easier to write programs in Python, although as it's an interpreted language, the source code is part of the installation package and may be exposed. It is also necessary to install the Python runtime as it is not currently preinstalled on any phones. [R1] discusses the pros and cons of using Python and Symbian C++ in detail.

3 Architecture for Multi-Runtime Projects with Widgets

As described in the first paper of this series [R1], the interface between the Web Runtime Widget and another runtime is a local network connection to a local Symbian C++ or Python HTTP server.

Figure 1 shows the architecture of such a multi-runtime application. The local switchboard server acts as a combination of simple proxy and cache and provides some local functionality.

All requests from the Widgets are issued to the local switchboard server and this server forwards the request to the remote Web Service, returns the cached content or handles the operation locally.

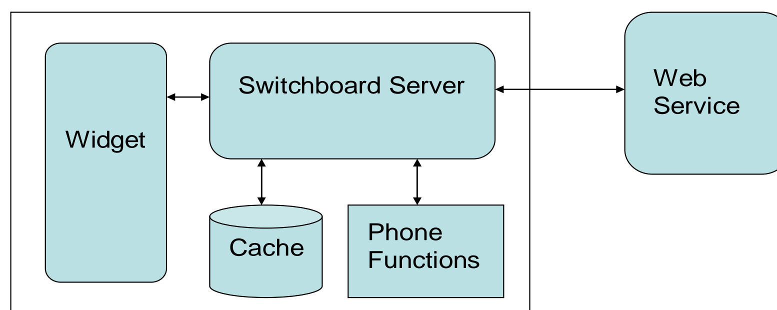

Figure 1: Architecture of a multi-runtime application

4 Web Services

Widgets are usually not designed as standalone applications and they use Internet connectivity to access functionality provided by a server offering a Web Service. To understand how this combination works and how to extend and support it with local functionality, it is necessary to look at the different types of Web Services and their suitability for use in multi-runtime projects.

4.1 The Architecture of Web Services

Web Services are basically an API for an application running on a server which is accessed over a network [R10]. In most cases the HTTP protocol is used and the data is encoded in XML, but the term Web Service does not refer to a specific technology. The W3C has a collection of standards and tools related to Web Services [R11].

There are different approaches to implementing Web Services. One is function centered and the other is more data centered. The following section takes a closer look at the current variants of Web Services, focusing on their suitability for the multi-runtime configuration.

4.1.1 RPC (Remote Procedure Calls) Web Services

Remote calling systems such as the programming-language-independent Common Object Requesting Broker Architecture (CORBA) [R12] and Remote Method Invocation (RMI) for Java are well established technologies for building distributed applications. CORBA uses an Interface Definition Language (IDL) to define the methods and data structures accessible over the network. Generators are used to create stubs for the target language to allow transparent access to the objects over the network.

These established RPC mechanisms like CORBA and RMI rely on servers exposed on particular IP ports to provide look-up services and method invocations. This works very well on a local, controlled network but can be troublesome over the Internet, where firewalls may block or modify traffic or redirect via a proxy service. To compensate for these limitations, RPC Web Services based on SOAP [R13] were developed, using HTTP as the transport medium. This method utilizes the same communication channels used by desktop browsers, allowing traffic to pass through firewalls and proxies. SOAP was the first widespread implementation of Web Services and it uses a similar concept to CORBA or Java RMI to provide functionality through remote procedure calls (RPC). SOAP defines its own XML format for data transfer and uses a XML format called Web Services Description Language (WSDL) to describe the interface of the Web Service.

There are various tools available for targeting SOAP services and many platforms are supported, including Java and .NET.

One important property of this concept when compared with the other techniques is that data is sent and received from just a few URLs, and functionality is implemented using XML files transferred to and from the server. Further information about this can be found in Section 4.2.1.

4.1.2 RESTful Web Services

The basic concept of RESTful Web Services is having resources that are identified by a global identifier. In order to manipulate these resources, the client and server communicate via a standardized interface and exchange representations of the resource [R14]. This is explained in more detail in Section 4.2.2.

Translated into more practical terms, this means that there are resources identified by URLs and they are manipulated using the HTTP methods `POST`, `GET`, `PUT` and `DELETE` to read, update, create and delete the representation of the resources (the actual documents containing the information).

The path in the URL is used to address the specific resource, for example, `http://..../Task/42/` to get the Task with the ID '42.' Using `GET` parameters such as `http://..../Tasks?id=42` should be avoided, and this is explained by [R14] in more detail.

For instance, to access a photo resource within a particular album it could be referenced as `http://www.service.com/user_x/album_spain/alhambra.jpg`, and the data returned would be in JPG format (much in the same way that a web browser retrieves an image). However, a RESTful API would also respond to a `DELETE` command which would remove the image from the browser or indeed a `DELETE` on the album `http://www.service.com/user_x/album_spain/` which might remove all of the images in the album.

The format of the resource is not restricted but in many cases XML or JSON [R15] is used. The content type is defined by the MIME type which is specified as an HTTP header.

In contrast to RPC Web Services, the RESTful Web Services are stateless: the server does not use a session to keep track of the clients. An application's state information is maintained using the representation of the resources: the documents. Taking the photo album example, this means that every single `GET`, `PUT` or `DELETE` operation is complete in itself and therefore they do not depend on each other. Any state information will be implicit in the data so, for example, the result of retrieving a picture will depend on the picture that was uploaded or how it was changed previously.

This approach reduces the server load and, very importantly, the network traffic because it allows very simple caching mechanisms in the client. Additionally, the server can use much simpler load balancing because no state has to be maintained by the server.

4.2 Implications for Multi-Runtime Projects

As described in Section 3, a multi-runtime project with Web Runtime Widgets isn't a point-to-point communication between two partners but actually between three. The local switchboard server acts as a middleman with the important role of reducing the communication over the external network connection as much as possible.

To achieve this optimization, the switchboard server has to be able to understand the communication in order to decide if a local operation should be performed, if data can be used from the local cache or if data has to be fetched from the Web Service. For this, the switchboard server does not need to understand the content of the messages but it has to participate in the transmission of the content.

With this in mind, let's take a look at different kinds of Web Services.

4.2.1 RPC Web Services

RPC Web Services use HTTP communication as a transport medium, but the operation to be performed, and the data necessary for that operation, is encoded in the XML that is sent to and received from the Web Service URL. For the switchboard server to do its work it has to decode the XML messages, save the extracted data and possibly even transform the data into a different format.

This requires the switchboard server to understand a lot more of the communication than is usually necessary to decide if a message might be cached, and for how long. Because the communication is not intended to be stateless, the switchboard server also has keep track of the state.

4.2.2 RESTful Web Services

RESTful Web Services use the HTTP layer to define the methods that are performed. `GET`, `PUT`, `POST` and `DELETE` are used to specify the kind of operations, the URL is used to qualify which data

is addressed and the HTTP header information is used to control the caching behavior. The HTTP error codes are used to return the basic results of an operation.

This means that the switchboard server only has to know about the typical HTTP transactions and does not need to look at any of the data transferred; it just needs to store it for caching and then pass it on.

These mechanisms make the RESTful Web Services ideal for multi-runtime projects that use Widgets. The design of the switchboard server can be kept quite simple and can use existing HTTP libraries to perform most of the required functionality.

5 RESTful Widget Design Considerations

This section discusses some of the design considerations for a multi-runtime application with a Widget front end, a local switchboard server and a RESTful Web Service.

5.1 The URLs

Seen from the Widget, all URLs point to the `localhost` in order for all requests to be handled by the local switchboard server, so it looks like this:

```
http://localhost/WebService/...
```

The local switchboard server has to do the mapping to the actual host which provides the Web Service and it must decide which services are handled locally.

The switchboard server listens on a certain port and ports other than the common port 80 or 8080 should be used to avoid conflicts with other applications.

5.2 Local or Remote

One way to distinguish between local and remote services is to use URLs to provide an indication, like this:

```
http://localhost/WebService/remote/...
http://localhost/WebService/local/...
```

This indicates to the developer of the Widget where any given request is handled. However, this might be useful for the developer but where the request is handled may well change during the project's lifetime. This would impact the Widget unnecessarily and so the best way to distinguish between local and remote services is to provide a simple URL to the Widget and let the local switchboard server do the mapping to the remote URLs.

5.3 Multiple Web Services

There is no design restriction on the number of Web Services which can be contacted by one application because the remote HTTP requests are performed by the switchboard server and there are no restrictions about which servers can be contacted for Symbian C++ or Python applications.

5.4 Access Control

An important requirement is that a user of a Web Service has to be identified and has to have a user name and a password to get access to the service. The easiest way to do this is through the HTTP authentication mechanism. The credentials can be passed to the server through the HTTP headers and the server responds with the HTTP error code 'forbidden' if authentication fails or it returns the data if successful.

6 An Example of a RESTful Widget

After a lot of theory in the last sections, it is now time to look at a RESTful Widget in practice. As an example we will look at how to implement a Widget with a corresponding Web Service for a janitor maintenance service.

6.1 The Use Cases

Before looking at the design, it's always a good idea to establish the use cases to ensure a shared understanding of the situation.

The janitor service is responsible for the maintenance of several buildings in different parts of the city. A call center takes the calls from residents reporting problems such as a dripping water tap or for scheduling regular maintenance work such as mowing the lawn. The janitor gets a list of tasks for the day and drives around to do the work and reports back the work he has done.

Network coverage is not available over the complete area and so the network should not be kept active all the time. This conserves battery power to make sure that the phone is available throughout the day for data exchange and for making phone calls if necessary.

In summary:

- the janitor gets a list of tasks for the day
- a Task informs him where he has to go and what task he has to perform
- after completing a job he reports what work he has done.

The following use cases would be quite useful for this service but will not be discussed in detail in this paper; they are included to give an idea of the benefits possible with a multi-runtime combination:

- a priority list is displayed for tasks that need immediate attention
- the janitor is informed of the nearest tasks to reduce the distance he drives.

The priority list can be implemented using a SMS push mechanism. The Web Service server would then send a SMS to inform the application on the phone that a priority task is available. The switchboard server would handle the SMS and sound an alarm to inform the user to check the new tasks.

6.2 Implementation

Before going into greater detail, we need to look at the basic techniques needed for the communication between the Widget, the switchboard server and the Web Service.

6.2.1 Widget

The basic information about implementing a Widget can be found at [R4], so this section will only cover the additional things necessary for the combination of different runtime environments.

To communicate with the Web Service, we use the JavaScript class `XMLHttpRequest` which can handle the different HTTP commands and can parse the received data as XML or as a flat buffer.

The first example shows how to issue a HTTP GET request to the local switchboard server.

```
var httpGetReq = false;

function loadDailyTasks() {
    httpGetReq = new XMLHttpRequest();
    httpGetReq.onreadystatechange = readyGetDailyTaskChanged();
    httpGetReq.open('GET', 'http://127.0.0.1:8888/JService/tasks/joe/');
    httpGetReq.send(null);
}

function readyGetDailyTaskChanged() {
    if ((httpGetReq.readyState == 4) && (httpGetReq.status == 200)) {
        refillTaskList();
    }
}

function refillTaskList() {
    var result = httpGetReq.responseXML;
    ....
}
```

First the object is created, then the target address is set, the callback function is provided and finally the request is executed. The callback function is called when the HTTP transaction state changes. The 4 indicates that the transaction has finished; the result is available if the result code is 200 (i.e., success).

The received data is parsed and the HTML table of the Widget view is updated.

The second example shows how a completed task is posted to the server.

```
var httpPostReq = false;

function postCompletedTask(task, task_id) {
    httpPostReq = new XMLHttpRequest();
    httpPostReq.onreadystatechange = readyPostCompletedTaskChanged();
    httpPostReq.open('POST', 'http://127.0.0.1:8888/JService/tasks/joe/' +
task_id);
    httpPostReq.send(task);
}

function readyPostCompletedTaskChanged() {
    if ((httpPostReq.readyState == 4) && (httpPostReq.status == 200)) {
    }
}
```

In this case, the ID of the task has to be appended to the URL because a single task is posted and not a list of tasks.

6.2.2 The switchboard server

This section gives a short overview of which classes and modules have to be used for implementing the switchboard server with Symbian C++ or Python.

The switchboard server has to be implemented as a part of the project and is not a ready-made component. Some of the application logic will be contained in this server, which makes it complicated to implement as a server for general use.

6.2.2.1 Implementation with Symbian C++

For implementation using Symbian C++, the system classes RHTTPSession and RHTTPTransaction are used to communicate with the Web Service and the simple HTTP server has to be implemented using a socket connection. The WebClient example, included in the S60 SDKs, illustrates how to use these classes.

6.2.2.2 Implementation with Python

The Python implementation utilizes the module httplib for different operations with the Web Service and the modules SimpleHTTPServer and BaseHTTPServer to implement the simple HTTP server.

Chapter 8 of [R7] explains in detail how to implement a web server and a web client with Python for S60.

The following example shows how to read the data from the Web Service:

```
import httplib

def getDailyTasks():
    conn = httplib.HTTPConnection("www.myserver.com")
    conn.request("GET", "/JService/Tasks/User/Joe/", buffer)
    conn.close()

    return buffer
```

The following example illustrates how to implement a HTTP server class and a handler class which handles GET and PUT requests.

```
import BaseHTTPServer, SimpleHTTPServer

class Server(BaseHTTPServer.HTTPServer):
    allow_reuse_address = True

class Handler(SimpleHTTPServer.SimpleHTTPRequestHandler):
    def do_POST(self):
        size = int(self.headers["Content-length"])
        buffer = self.rfile.read(size)
        process_post(self.path, buffer)

        self.send_response(200)
        self.end_headers()

    def do_GET(self):
        mime, reply = process_get(self.path)

        self.send_response(200)
        self.send_header("mime-type", mime)
        self.end_headers()

        self.wfile.write(reply)
```

The two classes Server and Handler, which are needed for implementing the server, derive from the base classes provided by the modules. The Handler class implements the two methods do_GET() and do_POST() defined by the Handler base class SimpleHTTPRequestHandler.

These two methods are called by the server class when the server gets a HTTP GET or POST request from a client.

The do_POST() method reads the posted data buffer and passes the buffer to the process_post() method. After the processing of the request, 200, the HTTP return code for 'OK,' is returned as the result.

The second method, do_GET(), calls the process method process_GET(), returning the MIME type of the result in the header and the data in the body of the response; it also returns 200 as the result code.

The following example shows a very simplified implementation of the methods for processing the POST and GET requests:

```
def process_get(path):
    mime, buffer = read_from_cache(path)
    if not buffer:
        mime, buffer = read_from_server(path)
        return mime, buffer

def process_post(path, buffer):
    try:
        post_to_server(path, buffer)
    except:
        queue_to_post_cache(path, buffer)

httpd = Server(('', 8888), Handler)
httpd.serve_forever()
```

The process_get() method retrieves the path of the URL. The cache is checked to see if the data is already there and, if not, it is read from the Web Service.

The process_post() method receives the path and the data buffer to be posted to the Web Service. First the post to the Web Service is executed and, if this fails, the post request is queued to the cache. The post_to_server() method has to check this queue before posting anything to the Web Service, to ensure that it posts queued posts first.

The main loop itself is quite simple. An instance of the previously-defined server is created with the information on which local address and port to listen on and the class which is used to handle the HTTP requests. After that the main loop is started.

6.2.3 Data format

The data is encoded as XML in this example because it can be handled by JavaScript easily.

6.2.3.1 Tasks of the day

The list of tasks for the janitor for the day is retrieved from the Web Service:

http://.../JService/Tasks/user/

```
<?xml version="1.0" encoding="UTF-8"?>
<TaskList>
    <Task id="42">
        <Name>Fix Dripping Tap</Name>
        <Address>Schulstr. 10, Mr. Miller</Address>
        <Description></Description>
    </Task>
```

```
        <Task id="43">
            <Name>Mow lawn</Name>
            <Address>Katreppeln 23a</Address>
            <Description>Weekly task of
                         mowing the lawn in the back yard.
                         Don't schedule this during lunch hours.
            </Description>
        </Task>
    </TaskList>
```

The data sent by the Web Service is a `TaskList` which contains a number of Task elements. The user name for which the tasks are requested is not provided as an query parameter after a '?' but as a part of the path. This illustrates the concept of the RESTful Web Service where the selection of the data is only done through the URL itself and not by `GET` parameters.

6.2.3.2 The completed task

If a task is complete, the information about this task is sent to the Web Service. The information can be cached locally and the switchboard server can keep a spool and post it when the network connection is available.

```
http://.../JService/Task/user/task_id/
```

```
<?xml version="1.0" encoding="UTF-8"?>
<Task id="42">
    <TimeStarted>20081111:123715</TimeStarted>
    <Duration>65</Duration>
    <DescriptionFix>Replaced the complete Tap.</DescriptionFix>
</Task>
```

The data posted to the Web Service is a single task which contains the information about the completion. Similar to the user name, the ID of the task is added to the path of the URL. Consequently, the posted data is now only a single Task item and not a list of tasks.

7 Integration

In the previous sections, we discussed the necessary components to combine a Widget and the switchboard server; this section describes how to install the multi-runtime package on the device.

For a good user experience, the multi-runtime application should be installed and started as a single application.

7.1 Starting the Application

The S60 5th Edition introduces the Platform Service API [R10] which enables Widgets to launch applications on the device. This API allows the Widget to start the switchboard server either using the UID in the case of a Symbian C++ application or by specifying the document and the MIME type in the case of a Python application.

For Widgets running on supported 3rd Edition devices this functionality is not available, so it is not possible to start the switchboard server from the Widget on these devices. Symbian C++ applications can use the browser control [R16] to run the Widget inside the application. In this case, the Symbian C++ application acts as a switchboard server and as the main application,

which is started from the application grid by the user and loads the Widgets files in the browser control.

Python doesn't support the browser control directly and so the user is required to start the Python switchboard server manually. To avoid this undesirable user experience it is necessary to implement a Symbian C++ extension for Python which uses the browser control to run the Widget.

7.2 Packaging

For the installation of this kind of multi-runtime application, a native SIS installation package has to be used that allows the installation of the binaries for the switchboard server and the Widget in one installation step.

The necessary signing depends on the capabilities used by the switchboard server. More information about signing can be found at [R17].

8 Security

The general security concerns are already discussed in the first paper of this series [R1].

For Web Runtime Widgets using a local switchboard server, the security mechanisms are quite easy. As HTTP is used as a transport medium, HTTPS can be used to encrypt the connection between the Web Service and the application on the phone. There is no implementation effort necessary to utilize HTTPS because it is available in the HTTP libraries for Symbian C++ or Python.

The local communication between the Widget and the switchboard server should use plain HTTP. This enables the switchboard server to read the transferred data if necessary and the security risk with an unencrypted local communication will be negligible in most cases.

9 Conclusion

There are several runtime environments with different programming languages available on Symbian OS. They all have advantages and disadvantages as described in [R1] and no single runtime that is ideal for all purposes.

To benefit from the advantages of different runtime environments, two or more runtimes can be combined in one project. Tools like KuneriLite [R2], Jarpa [R2] and Flyer [R3] provide support for integrating two runtimes and removes some of the complexity.

But even with the combination of multiple runtimes there is no single combination which is best suited for all purposes. This series of four papers has given an overview of some useful combinations and the supporting tools, to enable developers to choose the right combination for their own, particular project.

10 References

[R1]	'Multi-Language Programming: Part I - An Overview:' developer.symbian.com/main/downloads/papers/Multi_Language_Programming_Part1_Overview.pdf
[R2]	'Multi-Language Programming: Part II - Extending Flash Lite with Java ME or Symbian C++:' developer.symbian.com/main/downloads/papers/MLP_Paper+2_v1.0.pdf
[R3]	'Multi-Language Programming: Part III - Combining Python with Flash Lite and Symbian C++:' developer.symbian.com/main/downloads/papers/MLP_Paper_3_Python_Flash_Symbian C++.pdf
[R4]	Widgets: www.forum.nokia.com/main/resources/technologies/browsing/documentation/widgets.html
[R5]	5th Edition Platform Service API: www.forum.nokia.com/document/Web_Developers_Library/?content=GUID-46EABDC1-37CB-412A-ACAD-1A1A9466BB68.html
[R6]	Google Gears: gears.google.com/
[R7]	*Mobile Python: Rapid prototyping of applications on the mobile platform* by Jürgen Scheible. Book published by Symbian Press: developer.symbian.com/main/documentation/books/books_files/python/index.jsp
[R8]	PyS60: wiki.opensource.nokia.com/projects/PyS60
[R9]	PyS60 Community Edition: launchpad.net/pys60community
[R10]	Web Service: en.wikipedia.org/wiki/Web_service
[R11]	Web Services at W2C: www.w3.org/2002/ws/
[R12]	CORBA: www.corba.org/
[R13]	SOAP: en.wikipedia.org/wiki/SOAP_(protocol)
[R14]	RESTful Web Services: en.wikipedia.org/wiki/Representational_State_Transfer
[R15]	JSON: en.wikipedia.org/wiki/JSON
[R16]	Browser Control: wiki.forum.nokia.com/index.php/Browser_Control_API
[R17]	Symbian Signed: www.symbiansigned.com/

11 Author Profile

Bernd Wiegmann is the owner of WiB Software GmbH, a consulting company which specializes in the Symbian OS platform. He has many years of experience as a software architect for C++ and Java enterprise projects. He is an Accredited Symbian Developer and a Forum Nokia Champion.

Bernd's main focus is location-based applications for business and fun, and he is looking at new technologies that will make the development of mobile applications easier, helping to transform the mobile phone into a truly personal digital companion.