

1. Introduction

In the previous articles we covered:

- BlackBerry architecture
- Web service basics with Domino
- Basic BlackBerry MDS applications
- Advanced BlackBerry MDS applications
- Deploying BlackBerry Applications to handsets
- Introduction to JDE development

In this fourth article, we shall put all of the skills we have learned so far together, and show you how to get data from Lotus Domino.

A reasonable level of Lotus Domino and Java knowledge is assumed.

A word of warning. This article is long, comprehensive, and talks at length about complex communication issues. So get some coffee, forward your calls, and enjoy.

2. Transferring data with JDE Applications

Our first question is: Which method do you wish to use?

- Pull the data from the handset.
 - This is quite simple to understand and allows the user control over when data is transferred to and from our back-end data source. And it means that, since the handset will use standard Web calls, the handset need not be under our direct control.
 - Using this method, Jason Hook was able to distribute the Lotusphere 2007 Party application as a Web downloadable BlackBerry application, and have handsets transfer data to and from a public Web site. His application can be found at the <http://www.opencod.org> Web site. An interesting point about this application is that the BlackBerry smartphones that pulled the data from his public Web site were not attached to his BlackBerry Enterprise servers.
 - The downside of this approach is that you then have to rely on the user to synchronize the data on the handset with the data on the back-end database.
- Push the data to the handset.
 - The MDS-CS component of the BES server (installed by default) can push data to handsets. If the handset is not available, then the data is cached and lies in wait for the handset next to connect. In that respect, it's simple and reliable. You do, however, have to know the relevant MDS server for all of the users of your application.
 - You have to be able to push data to these BES servers, and therefore, it's implicit in this technique that these users have to be under your control. This is therefore more suitable for a corporate application, and less suitable for a public application.
 - The application itself must run all the time on the smartphone, as there's no guarantee as to when the update will arrive over the air.

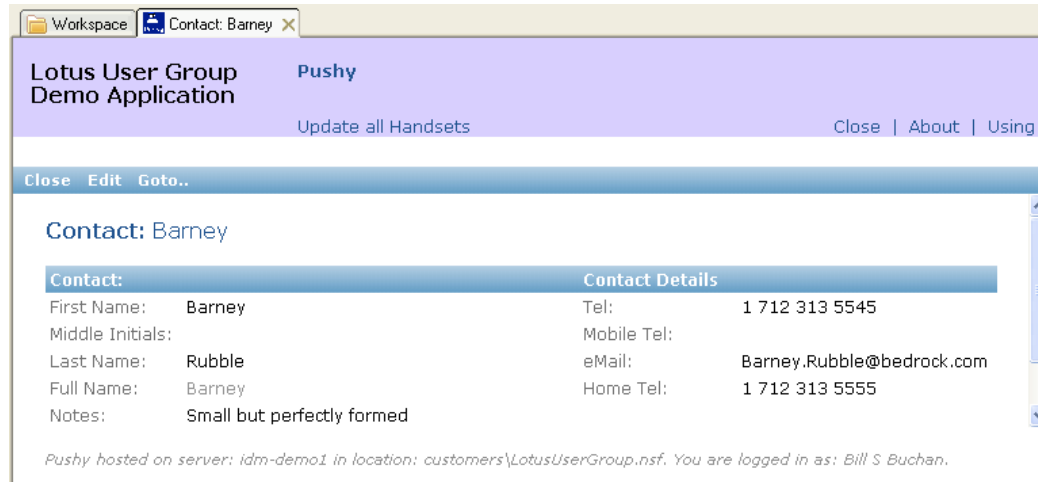
In both cases, we shall hold data off line on the handset. The JDE makes this very simple, indeed, and the data is held securely so that only applications we allow can see this data.

3. Our Lotus Domino Application

In this case, we shall use a very simple Domino application. We shall have a simple contact form, with various contact information fields on it. The application will have an action that creates the data push request and pushes it onto the BlackBerry server. I've called it "Pushy."

It's a really simple application. One form, three views, and a single Java agent.

The form looks like:



So far, so very, very simple.

4. Java Programming in Lotus Notes

This push request requires us to push an http request onto the BlackBerry server. Since LotusScript does not provide us with a low-level http protocol, we must use a Java agent to do this work for us.

If you've not done Java in Domino yet, I recommend Bob Balaban's book on this subject. It was published 10 years ago and is now out of print, but some folks sell their second-hand copies on Amazon.com.

Java has been in Notes for 10 years. The upside is that Java is probably supported on the Notes platform you're running right now. Unless, of course, you're running version 3. The downside is that the Java editor within Notes is pretty basic. Actually, I'm being kind. It's downright primitive. Thankfully, Maureen's team within Lotus is working on placing certain designer functions within Eclipse — so you can edit your Java within a nice grown-up Java IDE. If you can't wait for Maureen, search for Domiclipse — a fantastic (and free!) set of Eclipse-based tools from Keith Smilie.

If you're a LotusScript programmer — is it hard? Not really — the classes that you currently use in LotusScript are basically the same as the ones you can use in Java. You'll probably spend more time scratching around for string functions than figuring out the Notes Java interface.

5. Pushing Data to the BlackBerry Smartphone

In order to reduce the work necessary on the smartphone (always a consideration, as any portable device will have CPU and battery constraints compared to a server), we can simplify our data operation somewhat.

Instead of attempting to synchronize our data on the smartphone (a difficult and error prone piece of logic) we could instead just wipe all data from our application every time we have a data update. This operation is very simple indeed – however, it has the downside that all the data has to be transmitted every time. This is an important consideration, given that most users will be using very slow GPRS-style mobile connections.

Now, one feature of the BlackBerry mobile data system is that all data to every smartphone is compressed and encrypted – this makes very short work of text-based information, such as XML, and rather compensates for our slow data connection.

So, for applications with a low data overhead (in our case, a hundred or less contacts, for instance) this approach is desirable.

Should you wish to build an application which transmits thousands of records and/or is very frequently updated – say, more than a few times a day – then this approach may not be suitable. At this point you may consider building a synchronization engine. This is out of scope for this article.

6. The Lotus Notes “PushData” Java Agent

I’m going to don my hair-shirt and show it to you in the Lotus Domino Designer. First, when we create a new class, it inherits from AgentBase. So when we click on new agent, we end up with some template code, which looks like:

```
import lotus.domino.*;

public class JavaAgent extends AgentBase {

    public void NotesMain() {

        try {
            Session session = getSession();
            AgentContext agentContext = session.getAgentContext();

            // (Your code goes here)

        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Not hugely interesting so far. What we want to do is export our Notes document into a form that can be easily parsed. We could, of course, just create a text format, but why should we reinvent the wheel? Why not export this as XML?

```
private String getXMLFromDocument(Document d)
{
    String r = "";
    try
    {
        // Now break this document into an XML stream.
        String name = d.getItemValueString("Name.Full");
```

```

    r = "<CONTACT name=\"" + name + "\">" + "\n";

    r = r + "<NameFirst>" + d.getItemValueString("Name.First")
        + "</NameFirst>" + "\n";
    r = r + "<NameMI>" + d.getItemValueString("Name.MI")
        + "</NameMI>" + "\n";
    r = r + "<NameLast>" + d.getItemValueString("Name.Last")
        + "</NameLast>" + "\n";
    r = r + "<NameFull>" + d.getItemValueString("Name.Full")
        + "</NameFull>" + "\n";
    r = r + "<ContacteMail>"
        + d.getItemValueString("Contact.eMail")
        + "</ContacteMail>" + "\n";

    r = r + "<ContactTel>"
        + d.getItemValueString("Contact.Tel")
        + "</ContactTel>" + "\n";
    r = r + "<ContactMobile>"
        + d.getItemValueString("Contact.Mobile")
        + "</ContactMobile>" + "\n";
    r = r + "<ContactPersonal>"
        + d.getItemValueString("Contact.Personal")
        + "</ContactPersonal>" + "\n";
    r = r + "<ContactNotes>"
        + d.getItemValueString("Contact.Notes")
        + "</ContactNotes>" + "\n";

    r = r + "</CONTACT>" + "\n";
}
catch (Exception e)
{
    System.out.println(
        "Caught exception during document processing..");
    e.printStackTrace();
}
return r;
}

```

So, as you can see, we pass in a Notes Document and get back out a string of formatted XML. We could have used a DOM structure and got back XML that way, but this way is nice and simple, and fast. I wouldn't recommend it for a large and complex structure, but for a small number of fields, it's fine.

So far, so good. How do we then pass this to a BlackBerry smartphone?

One of the services on the BES server is the Mobile Data System (MDS) service. This has a handy store and forward service, which we can exploit. This guarantees that data we push onto the MDS service is delivered to the relevant smartphone. So how do we push data onto this service?

We simply put together an http protocol "Post" request and push it to a particular URL. In our case, for simplicity, we shall push it to one target address. This could be an individual smartphone, or it could be a group on the BES server.

The URL we should push to is of the form:

```

http://<hostname>/push?Destination=<device or group>&Port=<bbPort>
&REQUESTURI=/

```

So, we can easily pick up the hostname, BlackBerry Port number, and destination port address from a configuration document.

Bearing this in mind, we could then extend our "NotesMain" function to do the following:

```
import lotus.domino.*;
import java.io.*;
import java.net.*;
import java.net.URL;
import java.util.*;

// Create a new random number
private Random random= new Random();

public void NotesMain()
{
    try
    {
        Session session = getSession();
        AgentContext agentContext = session.getAgentContext();

        Database db = agentContext.getCurrentDatabase();
        Document docConfig = db.getProfileDocument(
            "Configuration", "");
        String mdsServer = docConfig.getItemValueString
            ("mds.server");
        int mdsPort = docConfig.getItemValueInteger
            ("mds.port");
        String bbGroup = docConfig.getItemValueString
            ("BlackBerry.Group");
        int bbPort = docConfig.getItemValueInteger
            ("BlackBerry.Port");

        System.out.println("BlackBerry Group: " + bbGroup
            + " and BlackBerry Port: " + bbPort);

        View vLookup = db.getView("XML");

        Document d = vLookup.getFirstDocument();

        URL url = new URL("http", mdsServer, mdsPort,
            "/push?DESTINATION=" + bbGroup +
            "&PORT=" + bbPort +
            "&REQUESTURI=/");

        // Construct a valid XML output stream from all our records
        String o =
            "<?xml version=\"1.0\" encoding=\"utf-8\"?><CONTACTCOLLECTION>";

        while (! (d == null))
        {
            o = o + this.getXMLFromDocument(d);
            d = vLookup.getNextDocument(d);
        }
        o = o + "</CONTACTCOLLECTION>";

        papPush(o, url, bbGroup, bbPort);
        vLookup = null;
    }
    catch(Exception e)
    {
        System.out.println(
            "Caught exception during document processing..");
        e.printStackTrace();
    }
}
```

That wasn't too hard. Just:

- Pick up this database
- Pick up a configuration document
- Compute our URL
- Prefix our XML string
- Iterate through all documents, adding in their XML
- Terminate our XML string
- Call the mysterious papPush function ...

So, what does papPush do? It takes in our target URL and our XML string and then constructs the relevant http session to the target server. It then pushes that http/XML request onto the server. This is what it looks like:

```
private void papPush(
    String output,
    URL url,
    String bbGroup,
    int bbPort)
{
    // The PushID is a unique number for this push request.
    String pushId = "pushID:" + random.nextInt();

    String errorCode = null;
    try
    {
        System.out.println(" sending PAP request to "
            + url.toString() + "; pushId = " + pushId);

        HttpURLConnection mdsConn = (HttpURLConnection)
            url.openConnection();
        String boundary = "";
        boundary = "asdlfkjiurwghasf";

        mdsConn.setRequestProperty(
            "Content-Type",
            "multipart/related; type=\"application/xml\"; boundary=" +
boundary);
        mdsConn.setRequestProperty("X-Wap-Application-Id", "/");
        mdsConn.setRequestProperty("X-Rim-Push-Dest-Port",
            "" + bbPort);
        mdsConn.setRequestMethod("POST");
        mdsConn.setAllowUserInteraction(false);
        mdsConn.setDoInput(true);
        mdsConn.setDoOutput(true);

        System.out.println(output);

        OutputStream outs = mdsConn.getOutputStream();
        copyStreams(new ByteArrayInputStream(output.getBytes()),
            outs);
        mdsConn.connect();
        ByteArrayOutputStream response = new
            ByteArrayOutputStream();
        copyStreams(mdsConn.getInputStream(), response);

        int httpCode = mdsConn.getResponseCode();
        if (httpCode != HttpURLConnection.HTTP_ACCEPTED)
        {
            throw new Exception("MDS returned HTTP status: "
```

```
        + httpCode);
    }
}
catch (Exception exception)
{
    if (errorCode == null)
    {
        errorCode = exception.getClass().getName();
    }
    System.out.println(" encountered error on submission: "
        + exception.toString());
}
}

public void copyStreams(
    InputStream ins, OutputStream outs) throws IOException
{
    int maxRead = 4096;
    byte [] buffer = new byte[4096];
    int bytesRead;
    for(;;)
    {
        bytesRead = ins.read(buffer);
        System.out.println(buffer);
        if (bytesRead <= 0) break;
        outs.write(buffer, 0, bytesRead);
    }
}
```

Whilst it might seem impenetrable, it's a fairly standard piece of Java boilerplate code.

Congratulations. We now have a skeleton Java push agent. Now let's discuss how the BlackBerry application is going to function.

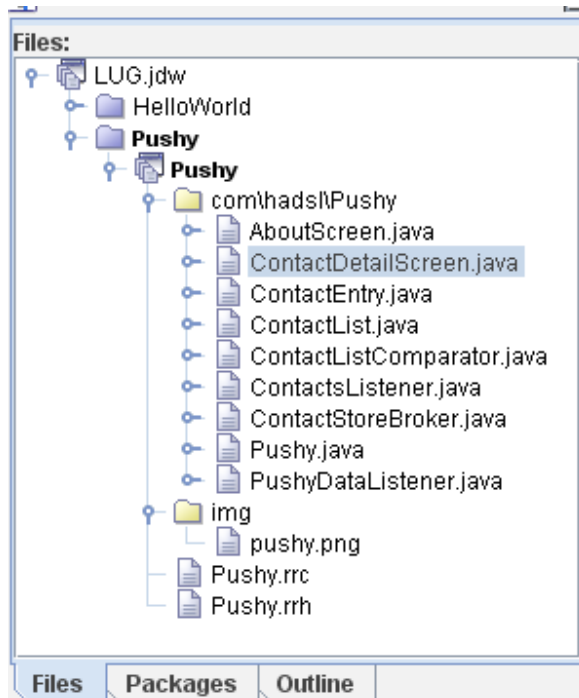
7. Creating the BlackBerry Application

Let's not focus too much on the basic BlackBerry application development — we covered that in the last article. Let's instead talk about how this application is structured.

We have:

- A main application class.
 - This has a function "main," which handles how the application should start up. More on how "main" is constructed in the next chapter.
 - It initialises the screen and shows a list of contacts.
 - Clicking on each contact will show more information on that contact.
 - Remember, the whole point of this application is to store this contact list on the BlackBerry smartphone secure memory so that it can be utilized offline.

- A contact class that holds all the information for that contact. Ideally, we'd like to store all data for this contact within this class, so we can easily expand this class later on without breaking the application.
- A collection class which holds zero or more contacts. Ideally, this collection class should be easy to use, and should sort the records upon insertion into some logical order. We'll also need a class that provides a comparison of two of these contact objects.
- A class which allows the display of that contact. We'd like to be able to select relevant fields and be able to call if that field is a phone number, etc.



- A class which deals with the incoming push data and updates our list on the BlackBerry smartphone secure memory store. This class should understand the format of the incoming data, and be able to parse it into a format understandable by this contact class.

8. Building an auto-run BlackBerry Application

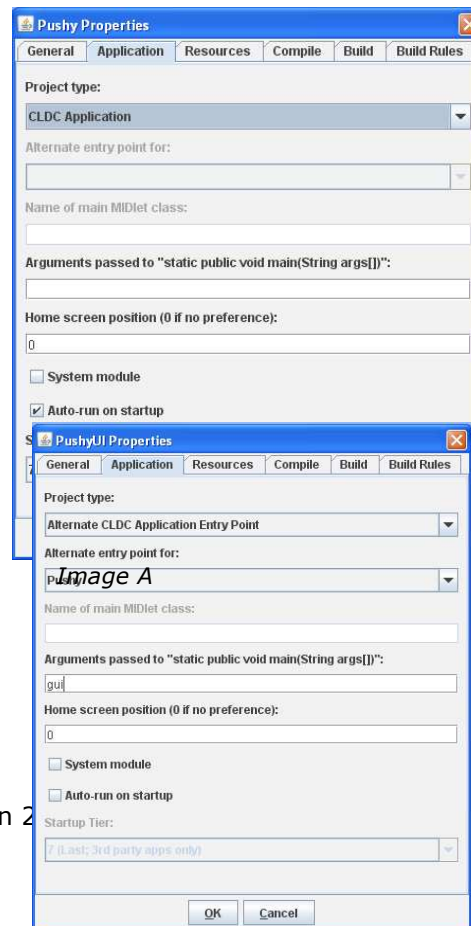
This is one of the areas in which I found the framework counterintuitive. In order for us to construct code on the smartphone that catches the push request, we must build an application which runs on the smartphone all the time and starts automatically on startup.

The application lies dormant until an incoming "push" data transfer is received, at which point its only function is to interpret this data and update our data store accordingly.

However, automatically started applications do not present a user interface, nor can they be interacted with.

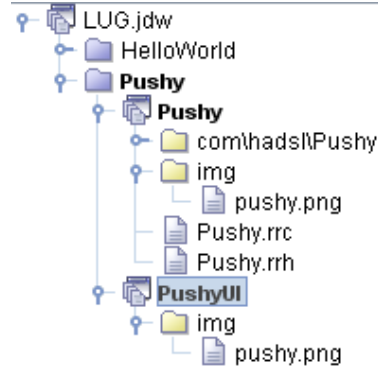
What you must now do is to:

- Set the project type to "auto-start," and set its level to 7 — third-party applications — as is shown on the project properties screen in image A on the right.
- Create a new project in the same workspace (I called it "PushyUI"), but set the type of the project to "Alternate CLDC Application Entry Point." You should then choose the main project to which this is



the secondary entry point. This second instance of the project can then be seen by the user and can possess an icon which will be placed on the user interface.

- Since we will inevitably have different startup routines for the two types of code, we can (as a rather ungraceful hack), set a textual parameter to our second project. We can pass a string — say “gui” — as a parameter on startup, and our “main” code can use this to distinguish between our automatically started code and our user initiated code.
- We should also add our image icon to the new stub project as before.
- As we are now running two separate programs on the smartphone, this can make it difficult to refresh the user interface, should the underlying data be changed by our automatically started, background program. Our workspace now looks like the one on the right.



Thankfully, both “applications” are bundled into the same distribution files, and therefore don’t increase complexity.

In our “main” constructor in our class “Pushy,” we can then decide whether this instance of the application is a background “catcher,” or a foreground UI class. This code looks like:

```
// Our Entry Point
public static void main(String[] args)
{
    try
    {
        if( args != null && args.length > 0)
        {
            System.out.println("User has initiated the application");
            thisInstance = new Pushy();
            thisInstance.enterEventDispatcher();
        }
        else
        {
            System.out.println(
                "Automatically starting the listener task");
            PushyListener.waitForSingleton().start();
        }
    }
    catch (Exception e)
    {
        System.err.println("An error has occurred " + e.toString());
        System.exit(0);
    }
}
```

9. The ContactEntry Class

Our ContactEntry class is our basic building block for our data. It’s very straightforward — just storing a number of attributes as a String.

```
public class ContactEntry implements
net.rim.device.api.util.Persistable
{
    // member variables
    private String name First;
```

```
private String _name_MI;
private String _name_Last;
private String _name_Full;

private String _contact_Tel;
private String _contact_Mobile;
private String _contact_eMail;
private String _contact_Personal;

private String _contact_Notes;

public ContactEntry()
{
}
...
// And various getters/setters...
...
}
```

10. The ContactList Class

Storing data in a secure manner on the BlackBerry smartphone is actually very straightforward. We have to:

- Create some sort of unique application-specific long integer, which we are fairly certain won't be used by other applications. A common convention is to select the full name of the class and convert that into a long integer using the IDE.
- Retrieve any existing collection class from the memory store using that key. If no existing collection class exists, create a new blank one.
- Pass the entire collection class to the secure memory store.

In other words, we use standard Java persistence in order to save our entire data collection.

In our case, we embed this functionality within the ContactList class and use a Vector object, which in turn contains one or more ContactEntry objects.

The class constructor retrieves the Vector (or creates a new Vector) from the memory store, and the commit function commits changes to the memory store.

```
class ContactList implements List, ListFieldCallback
{
    private static final long ESTORE_ID = 0xeca3fb94f3423abcL;
        // com.hadsl.Pushy
    private ContactListComparator _comparator;
    private PersistentObject _eStore;
    Vector _list;

    ContactList()
    {
        // System.err.println("ContactList::ContactList()");
        _eStore = PersistentStore.getPersistentObject( ESTORE_ID );

        try
        {
            _list = (Vector) _eStore.getContents();
        }
        catch (Exception e)
        {
            _list = null;
        }
    }
}
```

```
        System.err.println("Removing existing persistent memory
store - incompatible data storage");
    }
    try
    {
        if (_list == null)
        {
            _list = new Vector();

            _eStore.setContents(_list);
            _eStore.commit();
        }
    }
    catch (Exception e)
    {
        System.err.println("Failed to set eStore contents: " +
e.toString());
    }
    _comparator = new ContactListComparator();
}

private void commit()
{
    // System.err.println("ContactList::commit()");
    _eStore.commit();
}

// Other list handling functions omitted...
}
```

As you can see, it's very straightforward to store objects in the secure memory store. A word of warning, however: This area of memory is relatively small — up to 64mb — so whilst a few contacts or persistent information is a good use of this memory, attempting to store a million contact entries might not be. Be frugal with this resource.

11. The PushyListener Class

The PushyListener class is the class that creates a standalone thread that will listen to a "Push" message from the BES server. I refer to this type of code as a "catcher." Quite simply, it will:

- Find an instance of itself in memory. If it can't find an instance, it'll create a new thread for itself and attach itself to the listening port.
- Wait for an incoming Push request. When it finds one, it:
 - Detaches our payload from the request. Remember, this is an XML data stream.
 - If it finds a valid XML data stream, it:
 - Erases the currently held list of contacts in persistent memory.
 - Parses the XML and extracts out a list of contact entries. For each contact it finds, it creates a new ContactEntry object, and adds it to our list in persistent memory.

This particular routine will be the one you spend the most time debugging. Thankfully, the JDE IDE will allow you to set a breakpoint in the code (even though it's in a separate thread, etc.) and allow you to step through.

As you can see from the code, parsing XML is fairly straightforward. Horrible, but straightforward. We pick up a root node and parse each child node for a contact, then parse the child nodes of that for an attribute name and attribute value.

We then check to see if we understand the Attribute name, and if so, load the relevant property within the ContactEntry object with that value.

There is an argument that the parsing and the validation code for ContactEntry should live within ContactEntry itself — if it were more complex, or were extended to handle multiple object types — I would certainly agree. At this point, to keep things simple, I left the parsing code within the catcher itself.

12. Displaying the Contacts

Finally, we are nearing the point where we can display the contacts.

- Within the main Pushy class itself, we build a home screen for the application.
- On that home screen, we define a UI object that displays a list.
- When we open Pushy, we then load that list from our ContactList class.

One last thing we must do is intercept menu keypresses and scrollwheel clicks, and:

- Establish which ContactEntry object is highlighted.
- Open that ContactEntry object using a separate screen. In our case, we called the separate screen "ContactDetailScreen." This screen then makes repeated calls to the object, creates labels (from our Resources bundle), and displays contact information on a read-only basis.
- Lastly we draw a "close" button (and add a close menu event), which then closes the ContactDetailScreen object from the UI Screen stack and returns us to the main screen.

13. Testing

Testing this application is fairly straightforward. You must:

- Download and install the BlackBerry MDS simulator from the BlackBerry developers Web site.
- Run the MDS simulator.
- Use the JDE development environment simulator to run the application. Debug mode is particularly useful, as any writes to "System.out.println" are captured and written to the Debug log.
- Find out and use the PIN number of the BlackBerry simulator in order to direct push requests from Lotus Domino.
- Configure Lotus domino to use the MDS simulator as a target BES server, and the simulator smartphone as the target smartphone.
- Activate the Java agent to push requests. You should now see the MDS simulator report Push activity, and you can now debug the Simulator smartphone. I found it particularly useful to set breakpoints in the incoming Push catch code, which meant I could then step through the data-parsing.

14. A quick Note about Smartphone Operating Systems

It should be noted that not all smartphones are equal. Their Java Operating systems are not exactly the same. One distinction I found very useful was between that of an 8700 style (non-trackball) BlackBerry, and the newer generation of thumbwheel-based phones such as the Curve (the 8300 — my personal favourite), the Pearl (the 8100), and the WorldPhone (the 8800).

Older non-thumbwheel phones ship with operating system level v4.1, whilst trackball-based phones ship with OS level 4.2. Newer phones now appearing with WiFi capabilities may ship with OS level v4.3. How does this affect you?

The applications that the BlackBerry JDE produces are operating system specific. JDE v4.1 supports smartphones with operating system v4.1 and above. JDE v4.2 supports operating system v4.2 and above, and so forth.

This means from a practical point of view, you have to use the JDE that corresponds to the oldest operating system in your client base.

This may not necessarily restrict you only using old functions — for instance, not being able to use trackball functions (v4.2) because you have to use JDE v4.1. It means that you have to build your application using the oldest JDE and create a downloadable application from that — and then, if required, use a later JDE and add the later functionality. If this is required, you will have to install multiple versions of the JDE on your development machine(s) and be very precise in terms of source code control. This does not double the amount of work you have to do — but does increase the complexity of your debugging and testing process.

For example, should I build an application using JDE 4.2 and attempt to run it on a BlackBerry 8700 (which by default has operating system v4.1), it will fail. In my case, it returned the cryptic (and catch-all error) "Out of Memory." When it was rebuilt using JDE v4.1 (and I then commented out some code that was v4.2 specific — the trackball handler specifically) it ran perfectly well.

Hence the desire, should you run a mixed operating system BlackBerry user base, to upgrade the older smartphones to newer operating systems if possible.

15. Code Signing

In order to protect sensitive areas on the BlackBerry smartphone, we are required to sign applications that use certain restricted APIs. In our case, accessing the secure data store requires us to sign our application.

You should check your build output — if it contains a line such as:

```
Warning!: Reference to class:
net.rim.device.api.system.PersistentObject requires signing with key:
RIM Runtime API
```

then you are required to add signatures.

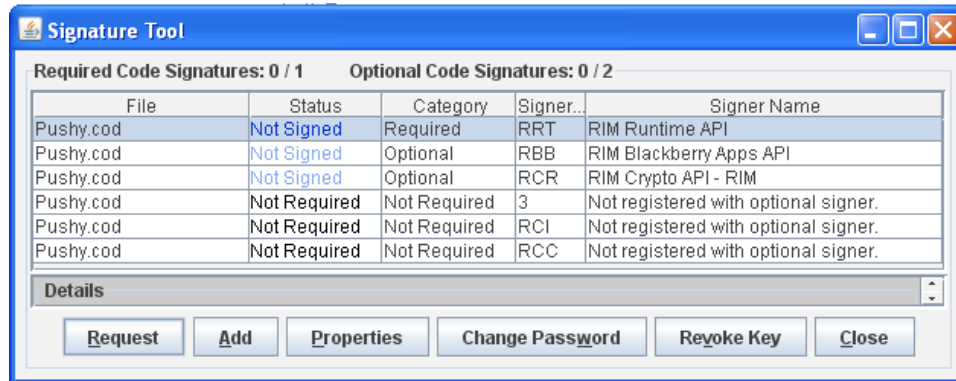
In order to sign an application, we are required to purchase (for a nominal \$100 or so) a "key" from Research In Motion — this can be purchased from their Web site but takes a week or so to deliver. Once you have this key, you can then insert it into the code-signing tool and sign your application once it has been built.

Whilst the application will, by default (this can be changed), run on the simulator without being signed — saving you the hassle of signing it every time — it will not run on smartphones without mandatory signatures.

To access the signature tool, click on the Build menu, and click on "Request Signatures."

The tool itself is very straightforward — it shows files in all active projects that require a signature. Click on the "Request" button, and the signature tool uses the key supplied by RIM to contact some RIM servers over the Internet and produce a new set of signatures for you, which are applied to the compiled application files.





16. Summary

This has been a whistle stop tour of building real-world applications that synchronize data between enterprise data systems and a BlackBerry smartphone. By using these techniques and the example application, you can easily create your own enterprise applications.

17. Resources

No BlackBerry Article would be complete without a trail of breadcrumbs to follow:

- <http://www.blackberry.com/developers> — the BlackBerry developer zone. Like all on-line resources, it's filled with lots of articles. You could (and may well) spend days in there.
- http://www.blackberry.com/knowledgecenterpublic/livelink.exe/fetch/2000/348583/796557/800451/1055819/What_Is_-_Sample_applications_demonstrating_BlackBerry_push_technology_-_Emergency_Contact_List.html?nodeid=1055822&vernum=2&cp=NLC-23 — This is an excellent "Push" application showing a Domino application pushing structured data to one or more handsets running a JDE listener task. A rare thing indeed — a well written demonstration application that works.
- "Professional BlackBerry" by Craig James Johnston and Richard Evers is one of the few books on the market dealing in significant depth with BlackBerry infrastructures and handsets. Well recommended.
- Programming Java on Domino 4.6 — Bob Balaban — is an excellent (if dated) reference on how to use Java with Lotus Notes.
- <http://www.Domiclipse.com> — Keith Smilie — is a framework for Eclipse, allowing you to edit Lotus Notes design objects.
- OpenCod is an open source resource for BlackBerry applications, in the same manner as OpenNTF is an open source Lotus Domino application repository. It can be found at <http://www.opencod.org>, and you are encouraged to both examine what is already available, as well as posting any relevant projects.
- The following white papers are required reading and are all available from the BlackBerry Developers Zone:
 - The BlackBerry Java Development Environment Development guide.
 - Deploying BlackBerry Applications.
 - Design Principles for BlackBerry Browser Applications.