Next, add the field for the description of the page and call it metaDescription. Finally, add the field for the keywords and call it metaKeywords. Neither field should be multi-value, because this will cause problems later in the HTML Header. Users can now control and enter search criteria for the most common search engines.

To translate this into an HTML Header, add the following code to your HTML Header:

```
"<meta http-equiv=\"Content-Type\" content=\"text/html;
charset=iso-8859-1\">"+@NewLine+
@If(metaRobots="1";"<META NAME=\"description\"
CONTENT=\""+metaDescription+"\">"+@NewLine;"")+
@If(metaRobots="1";"<META NAME=\"keywords\"
CONTENT=\""+metaKeywords+"\">"+@NewLine;"")+
"<META NAME=\"robots\"
CONTENT=\""+@If(metarobots="1";"index,follow,
noimageindex,noimageclick";"noindex,nofollow,
noimageindex,noimageclick")+"\">"+@NewLine
```

This code first checks the value of the field metaRobots. If an editor has enabled it, the META Tag "Robots" is added, which tells the search engines what to do and not do. "Index,Follow" means the pages will be searched for and added to their catalog, whereas "Follow" indicates underlying pages linked from this page will also be indexed. Using the keywords "NoIndex, NoFollow" ensures search engines skip the page.

When the editor has enabled the index option, the META tag "Description" and "Keywords" are filled accordingly with the entered content. Feel free to extend this example with more META Tags.

Keep it simple

Creating a straightforward content management system in Domino isn't that complicated, specifically because a lot of the core needs are already part of the standard package in Domino.

A content management system should be kept as simple as possible for the user. An overly complex system will slow down the process, and result in a slower delivery of content to the Web site.

I hope this article has given you a headstart on implementing content management at your organization. ■

Lotus Domino 5

PROGRAMMING

Java: The Easy Route to Server Add-In Tasks

By John Duggan

A little-known feature of Lotus Domino 5 is a task called "runjava" that runs Java server add-in tasks. This is used to run a server task called ISPY, which monitors the performance of delivering mail.

ISPY is written entirely in Java. Does this mean you can develop your own server add-in tasks in Java? The answer is yes, and this article outlines that process developing a simple add-in that monitors the share price of IBM stock and writes that value into a Notes document every few seconds. This Notes document could be your intranet home page, thus your own intranet can have almost real-time share prices!

Add-in tasks

Lotus Domino is a server that runs a collection of tasks that performs administration procedures, such as compacting databases, running agents, listening to client requests, updating indexes, etc.

An add-in is a customized program that runs alongside other tasks that make up the Domino server. Usually C or C++ developers write the customized tasks using the Notes API toolkit.

Most developer don't know they can write add-in tasks in Java. Java has many advantages over C/C++, including being easier to learn and support. Because Java has a large library of classes that can perform a wide range of tasks, it is quicker and simpler to develop add-in tasks in Java compared to C/C++.

Subscriber Download

- The source code required to write a simple add-in task.
- http://Advisor.com/Article/DUGGJ06

An ever better reason to use Java for add-in tasks is that it's cross-platform compatible. When I've compiled my Java program, I can copy the compiled program to a Windows and a Linux server and they'll work exactly the same, unlike a C or C++ version where I'd have to re-compile it for different operating systems.

Add-in tasks are predominately used for processing or performing a particular operation periodically on the server. Scheduled agents sound rather similar, but they have limitations on when they can run because they're dependent on the agent manager server task. Add-in tasks can perform periodic operations every minute or even every second. (There

John Duggan is an independent consultant and has been involved with IT development/management technologies since 1990. john_duggan@hotmail.com.

would have to be a very good reason to do something every second, though, to justify grinding the server to a halt!)

It's worth highlighting that an add-in task doesn't have to be periodic because you can have one that starts, performs a single operation, and exits, such as the "compact" task.

Server programming is sometimes seen as a black art. You'll see from this article that it's surprisingly straightforward to develop your own Domino server add-in task.

Write a simple add-in task

This article shows you how to develop a simple add-in task that reads a specific Web page at a predefined interval, extracts the share price, and updates a Notes document.

All the source code from this article is available for download at http://Advisor.com/Article/DUGGJ06. You can find the Java add-in source code in an agent called IntranetStockChecker.

I've stored the code within an agent for convenience, because I can compile the code in Notes Designer and export the compiled class to the hard drive using another agent called Extract compiled classes. This is useful because I'm not required to have a Java development tool, such as the Sun JDK or VisualAge, installed to create and compile Java programs when I can use the Notes designer.

For those of you interested in how this works, whenever Notes compiles Java code, it's packaged in a .JAR file called %%object.jar%%, which is stored as part of the agent's design. If you know the NoteID of the agent (which you can find from design properties), you can open the agent as a Notes document, get the embedded object, and extract the .JAR file.

Unfortunately, knowing the NoteID is rather cumbersome. Fortunately, all design elements are just documents, so you can modify a Notes view to show just Java agents. The extract agent reads each document in the view (remembering that the documents are agents) and extracts the .JAR file.

Using the add-in tasks require the class files, not the .JAR packages. In other words, you need a specific class file in the package. Fortunately, Java provides objects that read or unzip .JAR files and extract their contents. I haven't added code to intelligently extract files to the correct directory on the hard drive-I'll leave that project for you.

Write the add-in task

The first part of the add-in task is to include all the relevant Java classes you want to use:

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

More importantly, you're writing a server add-in, so you must add these lines:

```
// functions for java add-in tasks
import lotus.notes.add-ins.JavaServerAddin;

// functions for the message queue
import lotus.notes.internal.MessageQueue;
```

A message queue is a technology that lets Domino and Notes applications communicate with each other using "inter-process" communication. In other words, when you type in a command such as:

```
tell http restart
```

you're telling the Domino server to put the message "restart" into a message queue that belongs to the task called "http" which is the Domino Web server. The http task reads messages from its own queue and performs the relevant action. When it reaches the message "restart," it restarts itself.

It isn't mandatory to provide a message queue for add-in tasks, but it does let the task be more user-friendly because you can issue commands to it at the server console. If a task doesn't have a message queue, you can only start and finish it when the server shuts down.

Any class you develop for a server add-in must inherit JavaServerAddin. This class provides all the functionality you need to set up a basic add-in task. You can do this by using the "extends" keyword:

```
public class IntranetStock extends JavaServerAddin
```

There are a number of functions you must provide for the add-in to work correctly. The first is a "main" function that's activated when the "runjava" task calls the add-in. Your main function starts the add-in as a new thread.

```
static IntranetStock stockChecker;

public static void main(String[] argv)
{
  stockChecker = new IntranetStock() ;
  stockChecker.start();
}
```

When your object IntranetStock is created, you should assign it a name:

```
public void IntranetStockChecker()
{
setName("IntranetStock");
}
```

You also need a function to handle the shutdown. This is straightforward and should call the stopAddin in the JavaServerAdd-in class:

```
public void stopAddin()
{
  super.stopAddin();
}
```

For convenience, you'll provide a function that prints a message to the server console and the log file:

```
private void LogInfoMessage(String msg)
{
  AddinLogMessageText(ADD-IN_NAME + ": " + msg, NOER-
ROR);
  AddinIdle();
}
```

The next step is to write the code to do all the processing, which should be contained in a function called runNotes():

```
public void runNotes()
{
  .. all the processing of the add-in task takes place
here
}
```

When this routine is called, you'll use a function called "AddinCreateStatusLine" to create the status that appears when you enter a "show tasks" on the server.

```
handle = AddinCreateStatusLine(ADD-IN_NAME);
```

Remember, when you finish the add-in task, you have to call AddinDeleteStatusLine to de-allocate the status line.

At this stage, you can activate the message queue for your task. Every message queue will have a name, and in this case you've defined it as:

```
String msgqName = MSG_Q_PREFIX + "INTRANETSTOCK";
```

Using the above name, you can initialize and create the queue:

```
MessageQueue messageQueue = new MessageQueue();
int i = messageQueue.create(msgqName,20,0);
```

Depending on the type of task you create, you may only want one task running at a time. Thus, do a quick check to see if it's already running by checking if the queue already exists:

```
if (i == MQ_DUPLICATE_ERROR)
{
// task already running, stop add-in
  ..
}
```

At this stage, try to open the message queue. If it's already open, you can decide to stop the task or continue running without a message queue:

```
i = messageQueue.open(lemsg, 0);
if (i != 0)
{
// Queue already open
  ..
}
```

The queue is now created. The purpose of your add-in is to update a Notes document within a particular database, so you'll have to get hold of the document you want to update. For simplicity, I chose to use the first document in the All Documents view:

```
Session session = NotesFactory.createSession();
Database db = _
  session.getDatabase("", "JavaAddin.nsf", false);
View view = db.getView("($All)");
Document doc = view.getFirstDocument();
```

You've initialized all your data and can now update your status for the add-in task. You do this with a call to AddinSetStatusLine with the handle you obtained from the AddinCreateStatusLine:

```
  AddinSetStatusLine(handle, "Checking price every " _
    + duration + " seconds.");
```

This means the above text, with the name of the add-in task, appears in the task list when you type "show tasks" at the server console.

Because there can be many tasks running on the server at any one time, your task shouldn't stop any other task running. So, use a loop:

```
while (AddinRunning())
{
OSPreemptOccasionally();
.. do processing
}
```

The function AddinRunning returns a true or false depending on whether your add-in should still be running. Passing a "quit" to the task at the server's console sets this value to false, aborting the loop and letting the task finish.

The function OSPreemptOccasionally() causes the add-in to give up control of the processor so other tasks can run. This function is required due to the way the Windows operating system functions.

Alternatively, you can use a function called AddinIdle that gives up the control of the processor to the server task and receives control back when the server decides the add-in may proceed.

Your main loop shouldn't take more than one to five seconds to complete, to keep the Domino Server happy. If it takes longer, you should use a function called AddinShouldTerminate that checks for termination because the task may be required to finish at any time.

Within your loop, you want to retrieve the share price and update the Notes document. Because you want to read the Web site after a number of seconds have elapsed, you can use AddinSecondsHaveElapsed. If you're timing at minute intervals, you'd use AddinMinutesHaveElapsed instead:

```
if (AddinSecondsHaveElapsed(duration))
{
  price = getPrice();
  LogInfoMessage("Current Price: " + price);

  doc.replaceItemValue("SharePrice", price);
  doc.save(false,false);
}
```

Within your loop, you want to process any messages that may be sent to your task:

```
if(messageQueue != null && messageQueue.isQuitPending())
  break;
if (messageQueue.get(args,length, 0,0)==NOERROR &&
!(messageQueue.isQuitPending()));
{
\\ process commands
..
}
```

Alternatively, if your add-in task doesn't perform periodic operations and relies only on messages, you'd wait for a message to arrive, like this:

```
if (messageQueue.get(args,length,
messageQueue.MQ_WAIT_FOR_MSG,0)..
```

Your add-in task handles two commands: "duration" states how many seconds it should wait before checking the Web page, and "help" gives further information about the task. The code to handle the commands looks like this:

```
     if (s.equalsIgnoreCase("DURATION"))
{
  String sDuration = st.nextToken();
  duration = new Integer(sDuration).intValue();
  LogInfoMessage("Duration set to " + duration _
    + " seconds.");
  AddinSetStatusLine(handle, "Checking price every " _
    + duration + " seconds.");
}
else if (s.equalsIgnoreCase("HELP"))
{
LogInfoMessage("Commands help, duration, quit");
}
else
  LogInfoMessage("Unable to recognise command");
```

When the shutdown of the task is requested, you must provide a clean shutdown. So, you should close the message queue and de-allocate the status line:

```
if(messageQueue != null)
  messageQueue.close(0);

AddinSetStatusText("Shutdown");
LogInfoMessage("Shutdown");
AddinDeleteStatusLine(handle);
```

Read the share price

You've completed your add-in, now you have to write one more function to retrieve the share price from an Internet page. To keep the design of the add-in simple, read a page from Yahoo (http://finance.yahoo.com/q?s=ibm&d=e) that provides IBM's real-time share price.

HTTPConnection is a useful Java class that can read pages from the Internet and put the HTML into a string. You can parse the string and look for any distinct patterns that tell you where the share price is located. The drawback of this approach is when the design of the page changes, it is likely you won't be able to retrieve the price.

XML avoids this problem by separating the design and data. Thus, in a live production environment, if the XML data is available, make sure you read that instead.

The first part of the function is to follow good practice and declare some constants you can easily change without having to modify the code at a later date:

```
// The Web page containing IBM's share price.
String WEBPAGE
="http://finance.yahoo.com/q?s=ibm&d=e";
// The text that appears just before the share
price.
String STARTKEY ="<td nowrap><b>";
// The text that appear just after the share price.
String ENDKEY ="</b></td>";
```

The next step is to connect to the Web page using HttpURLConnection. First, declare a new URL object:

```
URL url = new URL(WEBPAGE);
```

Using the URL object, you can pass it to the HttpURLConnection and start reading the page:

```
HttpURLConnection connection
=(HttpURLConnection)url.openConnection();
connection.connect();

BufferedReader reader = new BufferedReader(new
InputStreamReader(connection.getInputStream()));
```

You'll read the page, line by line, until you come across the starting text you provided in your constants:

```
String price = null;
String line = null;

while (((line = reader.readLine()) != null) &&
price==null)
{
 int startpos = line.indexOf(STARTKEY);

  // Have you found a line that contains your
  // starting text?
  if (startpos>0)
  {
    // Yes, you've found a line that contains your
    // starting text.
    // Does it contain your finishing text?
    int endpos = line.lastIndexOf(ENDKEY);
    if (endpos>0)
      price = line.substring(startpos _
        + STARTKEY.length(),endpos);
  }
}
```

After all that, you now know the share price and can return IBM's share price to the add-in which will write to the console and update the value in your Notes document.

Use the add-in task

When you've written and compiled your Java add-in task, you can now execute it manually on the Domino server by typing in the following command at the server console:

```
Load runjava IntranetStock
```

Or, you can set it to run automatically when the server starts by adding the name of the task to the ServerTasks settings in the NOTES.INI file:

```
ServerTasks=http,runjava IntranetStock, smtp
```

Figure 1: **Title**—description.

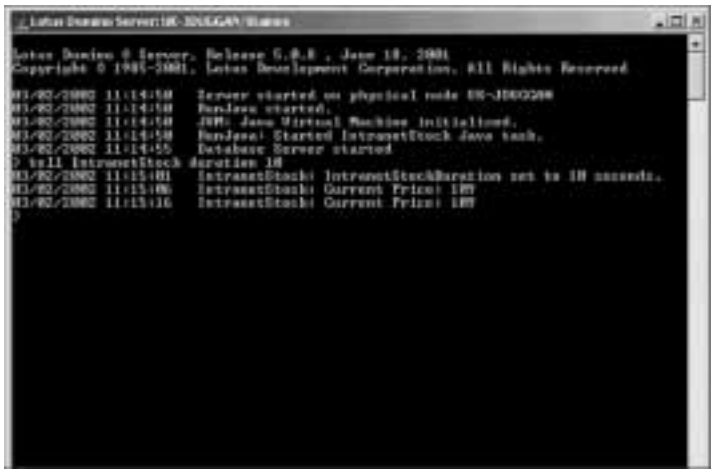

Figure 2: **Title**—description.

It's a Java program, so the 'runjava' task must start it. If you want to schedule the task for a fixed time, use the ServerTasksAt parameter:

```
ServerTasksAt1=runjava SharePrice
```

Check that the add-in task is running by typing "show tasks" at the server console. This shows all the server tasks currently running on the server along with their status. As shown in figure 1, you should see an entry called "IntranetStock" on the list.

Let the task run for a few seconds and you'll soon see messages, appearing on the console of IBM's current share price (figure 2).

If an exception occurs, check that your server has access to the Internet-or provide better error handling.

You can now issue commands to your add-in task, such as changing how often the task should check the price. For instance, say you'd like to change the task to check for the price every 10 seconds, so type the following at the server console:

```
Tell IntranetStock duration 10
```

A message appears confirming that it understood the command and the task will now check the price every 10 seconds.

Alternatively, you can create a Program document in the Domino Directory to run the task at scheduled intervals. This can be more convenient, because the program document is replicated across servers whereas the settings within the Notes.ini aren't replicated. The program document allows control of the task over a range of servers.

Finally, you can stop the add-in by typing the following at the console:

```
tell IntranetStock quit
```

Alternatively, you can tell the runjava task to close the task by typing in the following at the console:

```
tell runjava unload IntranetStock
```

If you'd like to see which Java tasks are currently running on the server, type the following at the server console:

```
tell runjava show tasks
```

Beware: Typing the above on pre v5.0.9 servers could crash the server.

Instant Java

Writing Java add-in tasks are easy and are considerably quicker to develop than their C/C++ counterparts. Although there are instances where C/C++ is a better choice than Java, such as performing low level functions and performance.

Listing 1 shows the complete list of functions available in the JavaServerAddin class, and listing 2 shows the complete list of functions available from the MessageQueue class. You can find further information about each of the functions in the class in the Notes API reference on http://www.lotus.com.

Listing 1: **Title**—Constants and function available from the JavaServerAddin class.

```
int NOERROR = 0;

short ST_UNIQUE = 0;
short ST_ADDITIVE = 1;

short VT_LONG = 0;
short VT_TEXT = 1;
short VT_TIMEDATE = 2;
short VT_NUMBER = 3;

String MSG_Q_PREFIX = "MQ$";


int AddinCreateStatusLine(String s);
boolean AddinDayHasElapsed();
void AddinDeleteStatusLine(int i);
boolean AddinIdle();
void AddinLogErrorText(String s, int i);
void AddinLogMessageText(String s, int i);
boolean AddinMinutesHaveElapsed(int i);
Vector AddinQueryDefaults();
boolean AddinShouldTerminate();
void AddinSetDefaults(int i, int j);
void AddinSetStatusText(String s);
void AddinSetStatusLine(int i, String s);
boolean AddinSecondsHaveElapsed(int i);


int CreateAndSendMailTrace(String s, String s1, String
s2, String s3);
String OSLoadString(int i);
void OSPreemptOccasionally();
void StatDelete(String s, String s1);
int StatUpdate(String s, String s1, short word0, short
word1, Object obj);
```

Listing 2: **Title**—Constants and functions available from the MessageQueue class.

```
int NOERROR = 0;

int MQSCAN_NOERROR = 0;
int MQSCAN_ABORT = 1;
int MQSCAN_DEQUEUE = 2;
int MQSCAN_DELETE = 3;

int MQ_WAIT_FOR_MSG = 1;
int MQ_OVERRIDE_QUIT = 2;
int MQ_SUPPRESS_DUPLICATES = 1;
int MQ_CLOSE_ONLYIFEMPTY = 1;

int create(String s, int i, int j)
int close(int i)
int get(StringBuffer stringbuffer, int i, int j, int k)
int getCount()
int getRefCount()
int open(String s, int i)
int put(int i, String s, int j, int k, int l)
void putQuitMsg()
int scan(StringBuffer stringbuffer, int i, int j,
MessageQueueCBObject messagequeuecbobject)
boolean isQuitPending()
```

Java power

Even if you're primarily a LotusScript programmer, this article will hopefully given yourself some idea of the power of Java that isn't available within LotusScript. After you've picked up the basics, you'll be wondering what all the fuss is about with Java and could be producing server add-in tasks very quickly impressing your boss! (.....if you have a boss!) ■