NLA Systems Ltd.

Digital Document Distribution

User and Administrator Guide

CYBERPOST2 – DIGITAL DOCUMENT DISTRIBUTION

User and Administration Guide

Table of Contents

INTRODUCTION TO CYBERPOST2 2

THE USER INTERFACE / USER GUIDE 4

SCREEN LAYOUT 5
OPERATION 6
Individual recipient selection 6
Document Parameters 7
User-Defined Filters 7
Activity Log Window & Progress Bar 7
Activity Log File 8
Attach File 8
Connect To Database 8
Emailing Documents 8
Faxing Documents 8
Preview 9
Printing 9
Robot 9
Online Update 9
Update Launcher Classpath 9
SETUP & OPTIONS 10
Details 10
Logo 10
Database 11
Mail Servers / Robot 12
Fax 13
User Defined Parameters 14

THE CPXML EXPRESSIONS 15

BUILDING THE CPXML INTERFACE 18

PROPERTIES 19
SQL SECTION 22
FIELD SECTION 26
REPORT PARAMETERS 31
CUSTOM METHOD INVOCATION 32

ZIGINEEZE. THE DYNAMIC DOCUMENT ROBOT. 36

ZIGINEEZE REQUEST XML 36
ZIGINEEZE REQUEST EXAMPLE 1 38
ZIGINEEZE REQUEST EXAMPLE 2 39

1

Introduction to cyberPOST2

cyberPOST2 interfaces with user-defined document templates and a data source to enable us to distribute dynamic documents via email, fax or SMS. We can also print these dynamic documents or write them to disk. The application also features a document dispatch robot called Zigineeze.

We can define our document templates in the form of HTML pages, text files or by using Jasper Reports, Crystal Reports and other popular report writers. To interface cyberPOST2 to our document templates we need to create a simple XML file. We do not need to be XML experts to create this file. All we have to do is to copy an existing one to a new filename and use it as a skeleton to compose the new one. This XML interface should have a .cpxml extension.

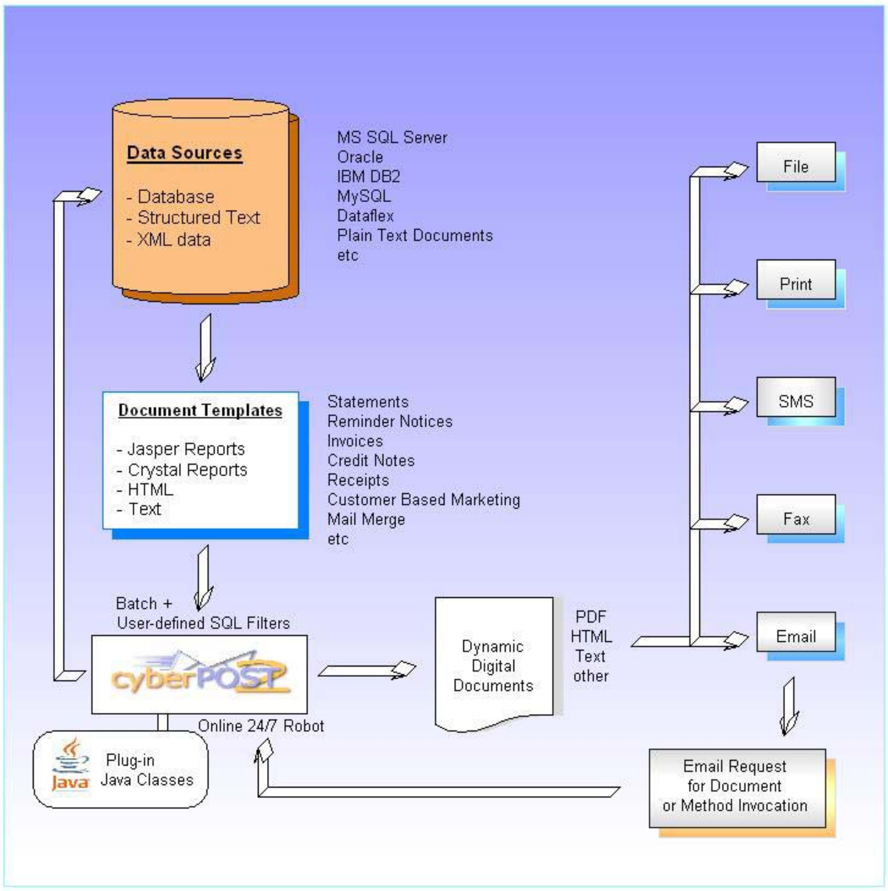


Figure 1 - cyberPOST2 architecture

For brevity we will refer to these files as cpxml files. For our intents and purposes we will refer to the combination of a document template and its cpxml file as a Document Definition.

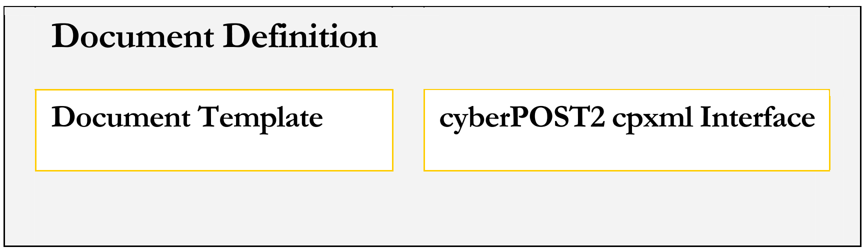


Figure 2 – Document Definition

The cpxml file provides cyberPOST2 with details about our templates and our data sources. For example we need to tell cyberPOST2 what parameters Jasper Reports or Crystal Reports are expecting. Additionally we need to specify to cyberPOST2 which SQL statements to execute on specific actions or events.

New to XML?
XML was designed to describe data and to focus on what data is. A quick tutorial is available here
http://www.w3schools.com/xml/default.asp

Before we go on to describe the XML elements involved to build a valid cpxml, we need to first describe the cyberPOST2 user-interface. This is described in the next chapter.

Note to the administrator

The next chapter also houses the user guide. This has been done because some arguments are tightly related and certain procedures cannot be strictly assigned to the user or the administrator.

The user interface / user guide

The user interface consists of a number of fixed ‘fields’. Each of these has a unique ID tag. All the fields available, together with their respective unique IDs are shown in Figure 3

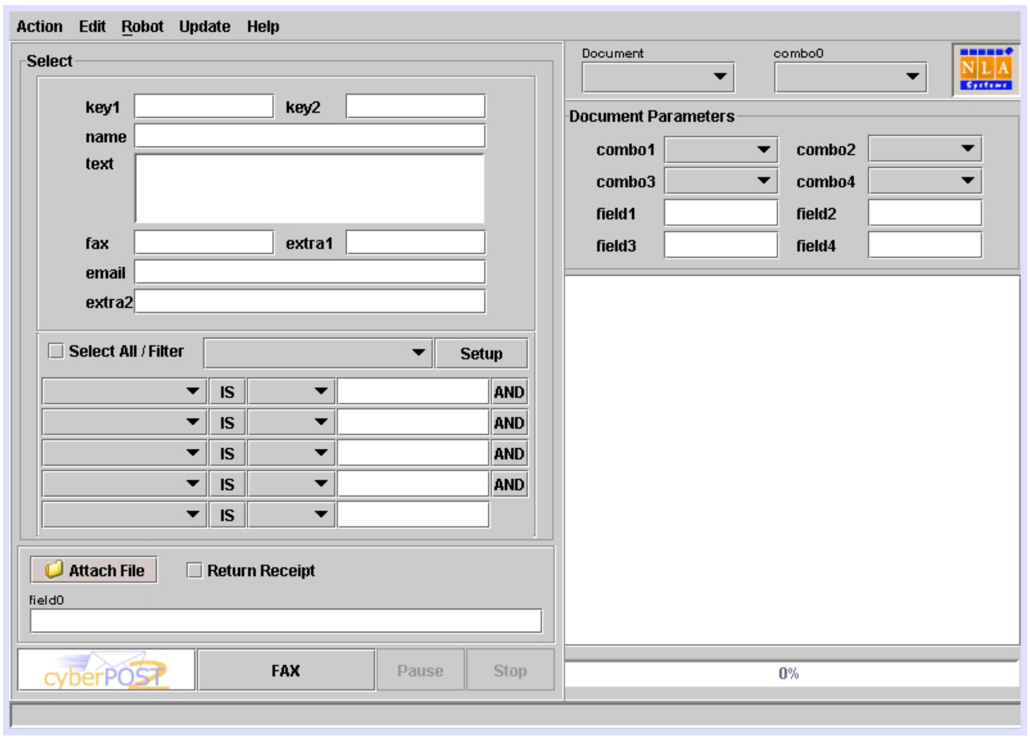


Figure 3 - The cyberPOST2 user-interface

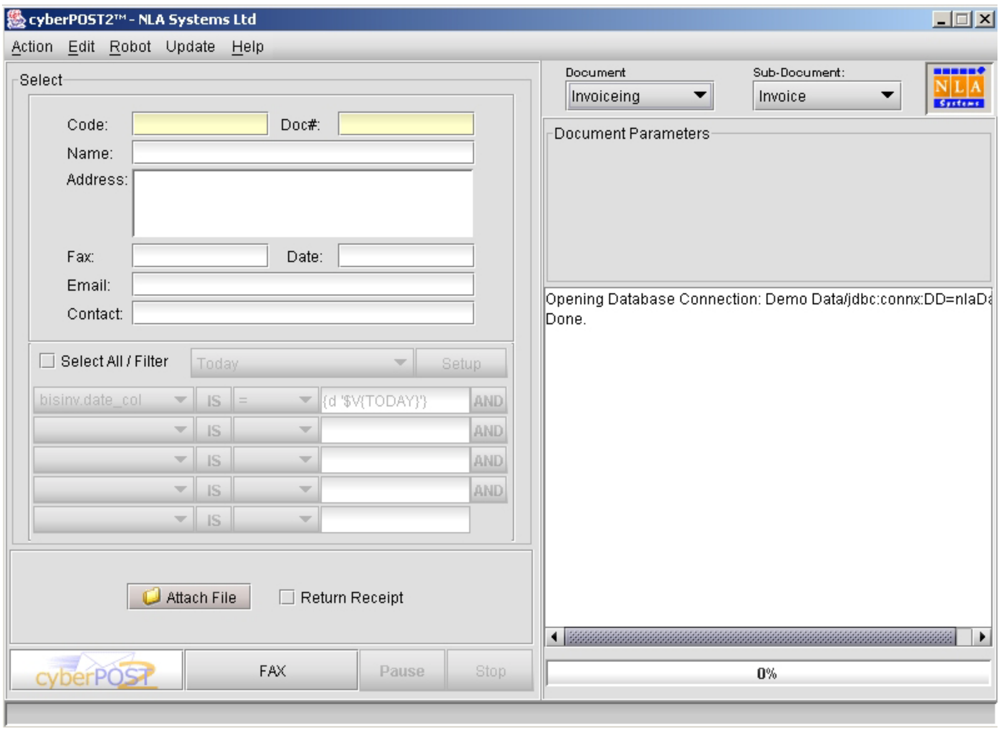
We would need any of these fields to provide user-input for our various document delivery applications. For example, in our sample application that distributes debtor statements, we can use field ‘key1’ as an input for account number, field ‘name’ for account name, ‘text’ for the address etc. Through the cpxml, we will bind these fields to data items in our data source. We will, also through the cpxml, assign a meaningful name to these fields for the actual user to see.

The ‘Document’ combo box found in the top-right area of the screen contains a list of available Document Definitions. The user switches from one document definition to another by selecting its name in this combo box.

We can add a Document Definition to cyberPOST2 by including its template and cpxml file in the 'reports' folder in the working directory. This is usually c:\cyberPOST2\reports in Windows systems and <user-home>/cyberPOST2/reports in Linux systems. When creating a new cpxml file it is good practice to copy an existing one to a new filename and using it as a skeleton to compose the new one.

Screen Layout

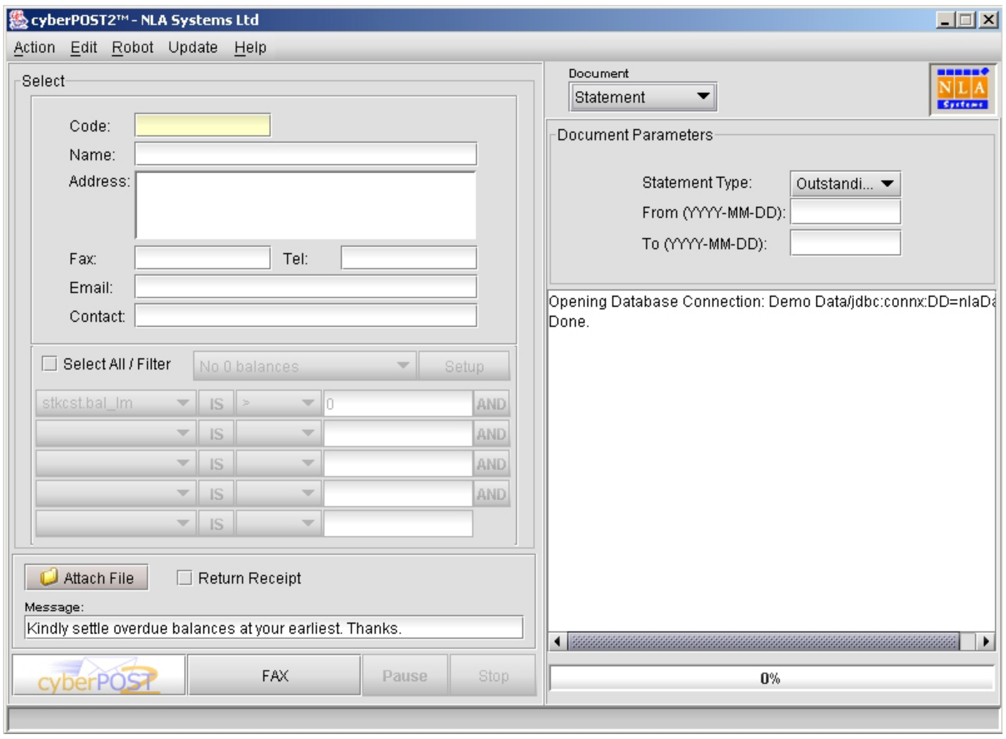
Screen layout changes inter-actively upon the choice of Document Definition. This is selected through the Document Combo Box. If a particular document has sub-documents, a second combo box shows up next to the Document Combo Box. An example of this can be seen in the demo applications. Document Definition "Invoicing" contains "Invoice", "Cash Sale", "Credit Note" and "Cash Sale Return" as sub-documents.



Document: Invoiceing
Sub-Document: Invoice

```
Opening Database Connection: Demo Data/jdbc:connx:DD=nlaD
Done.
```

Figure 4 – Invoicing sample application GUI



Statement Type: Outstandi...
From (YYYY-MM-DD):
To (YYYY-MM-DD):

Opening Database Connection: Demo Data/jdbc:connx:DD=nlaDa
Done.

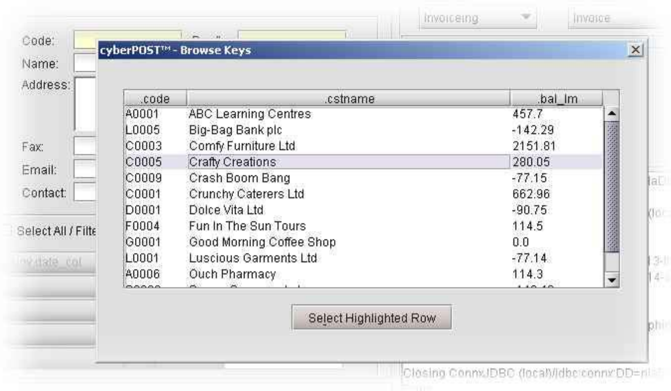
Message:
Kindly settle overdue balances at your earliest. Thanks.

Figure 5 – Statements sample application GUI

Operation

Individual recipient selection

Records can be selected by their unique keys or by browsing. A field that can be browsed has a light yellow background. If we know the key then we just enter it and press <ENTER>. If we want to browse, we DOUBLE CLICK in this field.



.code	.cstname	.bal_lm
A0001	ABC Learning Centres	457.7
L0005	Big-Bag Bank plc	-142.29
C0003	Comfy Furniture Ltd	2151.81
C0005	Crafty Creations	280.05
C0009	Crash Boom Bang	-77.15
C0001	Crunchy Caterers Ltd	662.96
D0001	Dolce Vita Ltd	-90.75
F0004	Fun In The Sun Tours	114.5
G0001	Good Morning Coffee Shop	0.0
L0001	Luscious Garments Ltd	-77.14
A0006	Ouch Pharmacy	114.3

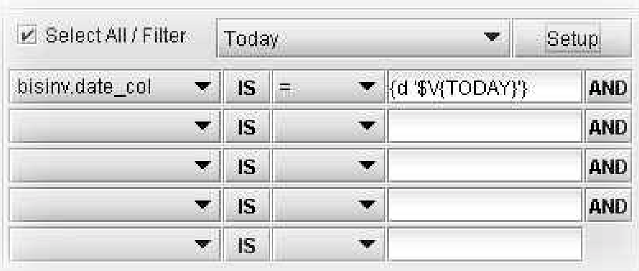
Figure 6 – Record Browser

Document Parameters

If the current document has certain options or parameters these can be entered in their proper field in the Document Parameters panel on the right hand side of the GUI. As an example in our statements sample application, we can issue statements as "Outstanding" or as a "Full Account". We can also request to show only those transactions that fall in a particular date range. Here we can enter a range of dates through the "From" and "To" fields.

User-Defined Filters

When sending documents by batch, unless otherwise specified by the user-defined filters, the system will send the currently selected document to all the recipients in the master table. In real life, this may rarely be the case. For example, in our statements application, it would not make sense to send statements to those debtors whose balance is 0. In our invoicing application we might want to send out all of today's invoices or we might want to send them weekly or even monthly. All this can be done through the user-defined array of filters consisting of an array of easy-to-use drop down list combo boxes, toggle buttons and text boxes. Filter settings can be saved under a suitable name for future use. These saved filter settings can be selected through the filter selection combo box. When more than one value needs to be inputted in the value text box such as in "Range" or "One Of", we should separate these values by a comma (,). The value text boxes can also contain cpxml expressions. For example, in the invoicing sample application we have included a filter to send today's invoices. In this case, instead of inputting each day's date in the value text box, we inserted the built-in cpxml variable $V{TODAY} in the form of an SQL date function {d $V{TODAY}}. We have saved this filter set under the name "Today".



✓ Select All / Filter	Today			Setup
bisinv.date_col	IS	=	{d '$V{TODAY}'}	AND
	IS			AND
	IS			AND
	IS			AND
	IS			

Activity Log Window & Progress Bar

The large scroll pane on the right hand side of the cyberPOST2 GUI displays system activity as well as any error messages. It also enables us to not only see the current message but also to scroll through the whole array of messages generated since cyberPOST2 was launched. This text is not lost when we exit cyberPOST2, but is saved in a log file for eventual viewing.

The progress bar shows what percentage of the current process has been completed.

Activity Log File

The system activity log mentioned above is simultaneously written to a file. Each line item gets time stamped. This file is located in the cyberPOST2 working directory. This can be viewed any time by clicking 'Action' ---> 'View Activity Log'. A separate file for every year is kept. We can view the log file for the current year only. We can view the log file for previous years, by opening the respective files in our favorite text editor or browser.

Attach File

We can also have an attached file that we can distribute along with the regular document. We click the 'Attach File' button to open the file browser and choose the file we want to attach. A message informing us that this file has been attached should appear in the scroll pane on the right.

To cancel the file attachment we can click the 'Attach File' button again and then the 'Cancel' button in the file browser. The message 'File to attach removed' should appear on the right.

Connect To Database

When cyberPOST2 is launched, it automatically attempts to connect to the database that has been defined in 'Setup & Options'. If for some reason or other, this database was not available when the application was launched, the 'Connect To Database' function enables us to do so once the database is available again. This function is found under the 'Action' menu bar item or can be invoked directly by pressing Control-D.

Emailing Documents

We click the cyberPOST2 button to email out documents according to what we have defined in the user-defined filters described above. Alternatively, we can access this function through the menu bar by clicking 'Action' -----> 'cyberPOST' or else, directly by pressing Control-C.

Any time while documents are being emailed, we can pause the process and later resume by pressing the 'Pause' button. We can also completely stop the process by clicking the 'Stop' button. *Note:* After the Stop button is pressed cyberPOST completes the sending of the current record before stopping.

Faxing Documents

We click the FAX button to fax out documents that do not have an email address but do have a fax number. Alternatively, we can access this function through the menu bar by clicking 'Action' -----> 'Fax' or else, directly by pressing Control-F. Here the user-defined filters act just like in the email process. The pause and stop buttons also work in the same manner as described in the email process. Fax status is continuously displayed in status message line at the bottom of the GUI.

Preview
We can preview a document to see how it will look like when received by the recipient. We can find this function under 'Action' in the menu bar. We can also go directly to it by pressing Control-V.

Printing
Those accounts that do not have an email address or a fax number can have their documents printed. This is accessible through 'Action' ----> 'Print To Default Printer' in the menu bar. We can also call this function directly by pressing Control-P. Same controls in the email and fax processes apply here as well.

Robot
To enable the robot, we can follow the instructions in Setup & Options - *Mail Servers / Robot* of this guide. When the robot is enabled, we should see the message 'ROBOT: Checking for incoming requests ...' in the status field from time to time. We can define this time interval. We should consider that the more frequent this check is set to occur, the greater the demand is placed on our computer and Internet connection. This should not be an issue if we have a powerful computer and a broadband connection. The default value is 60 seconds.

If the 'From' email address of any email request is not found in our database, cyberPOST2 replies to this requester with an error and asks him or her to phone us to amend his or her email address.

If for some reason the robot encounters problems such as a lost connection to the Internet, an error message is shown in the status field. As soon as the problem is rectified, this error message disappears.

We can temporarily switch on and off the robot through the "Robot' menu bar item. This applies only for the current session. When the application is launched again the robot's state will initialize according to what was specified in the setup.

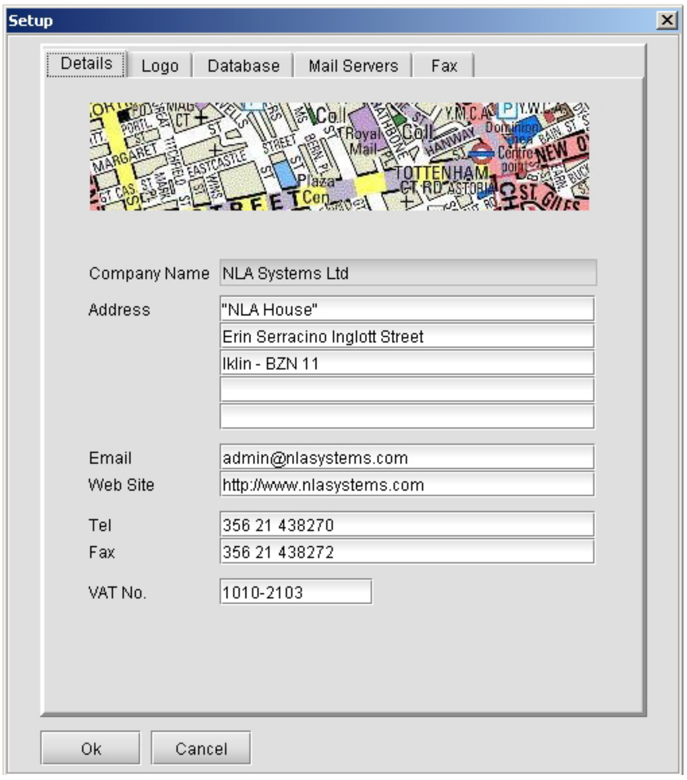
Online Update
The online update feature is an efficient way of keeping our application and its auxiliary files up-to-date. Rather than downloading the whole package every time an update is available, it compares our installed package against the online latest version and downloads only the new package components or files. This function can be accessed through the "Update" menu bar item or by pressing control-U.

Update Launcher Classpath
If we need to add our own custom jar files to the cyberPOST2 working directory such as a vendor specific JDBC driver, we need to include these in the application's classpath in order for them to be accessible. This is easily achieved by executing "Update Launcher Classpath" found under the "Update" menu bar item. Every time this function is executed, it gets a new list of jar files in the cyberPOST2 working directory and refreshes the classpath in the application launcher. The new jar files will be integrated on the next time we launch cyberPOST2.

Compiled classes that are not packaged in a jar file should be saved in the “cpCustom” folder that can be found in the “reports” folder in the working directory. Typically these classes are custom methods used by our report definitions called from the reporting engine such as Jasper or other custom methods to be invoked by the robot. When we add classes here, we do not need to update the classpath.

Setup & Options

The setup function is found under the 'Edit' menu bar item. These settings are divided into four tabs, namely Details, Database, Mail Servers and Fax.



Company Name	NLA Systems Ltd
Address	"NLA House"
	Erin Serracino Inglott Street
	Iklin - BZN 11
Email	admin@nlasystems.com
Web Site	http://www.nlasystems.com
Tel	356 21 438270
Fax	356 21 438272
VAT No.	1010-2103

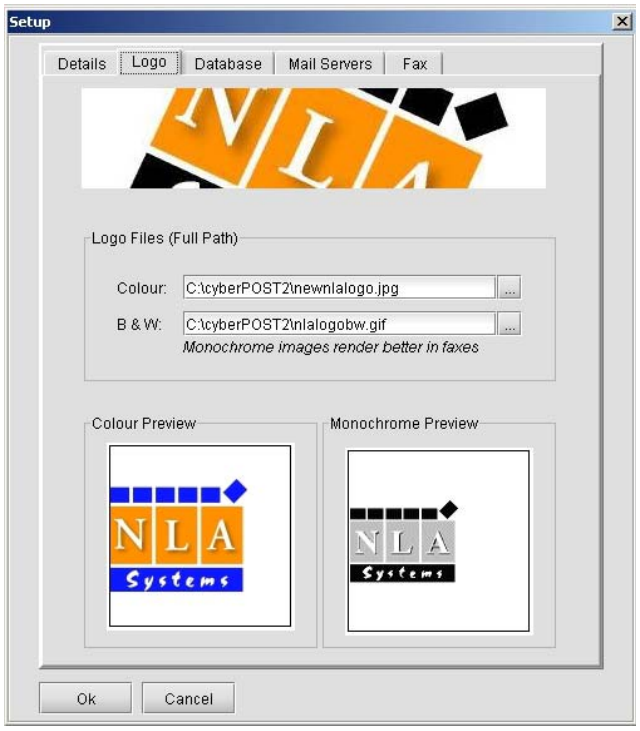
Details

Here we can enter our company's address, telephone, fax and VAT numbers, our contact email address and a URL to our web-site.

All these details are automatically passed as parameters to our report definitions. These can be used for example in the document header. In the case of a Jasper report definition, all these variables are automatically packed in the java.util.HashMap that is expected by the Jasper engine. We can also use them as cpxml variables in our text or HTML templates. By making use of these variables, all the changes edited in this form, are echoed through all our report definitions and templates. The variable names relating to each of these fields are detailed in a table found in Chapter 3.

Logo

For the same advantages cited in “details” above, the company logo is specified once in this little form and is then passed as a parameter to the reports just like the company details. If we are to use the fax process we should also specify a monochrome version of the logo if the color logo appears smudged in the faxed document. The filenames should be entered complete with their full path. Clicking the little button beside each field will open the file browser. As soon as each image



Logo Files (Full Path)

Colour: C:\cyberPOST2\newnlalogo.jpg

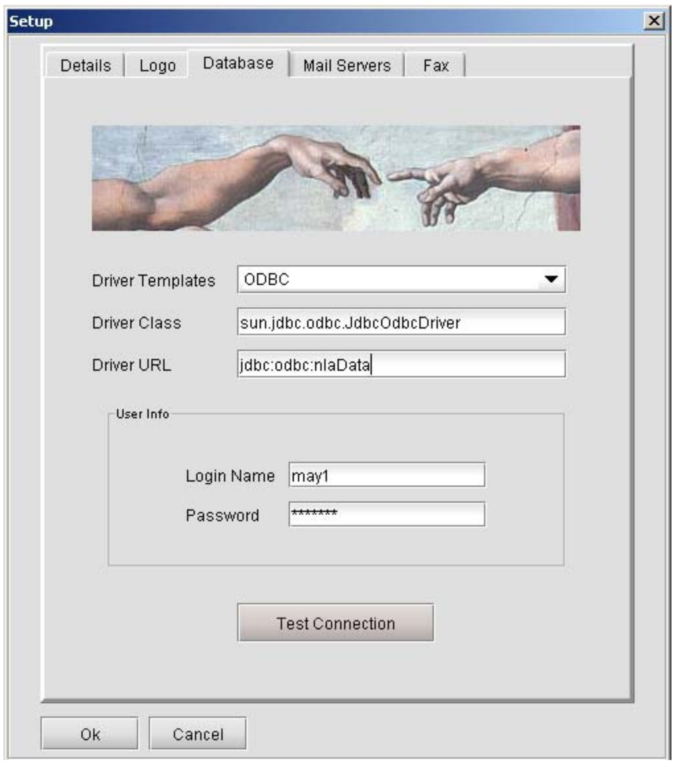
B & W: C:\cyberPOST2\nlalogobw.gif

Monochrome images render better in faxes

Colour Preview

Monochrome Preview

file name is entered, the little preview window assures us that we chose the intended image.



Driver Templates: ODBC

Driver Class: sun.jdbc.odbc.JdbcOdbcDriver

Driver URL: jdbc:odbc:nlaData

User Info

Login Name: may1

Password: *******

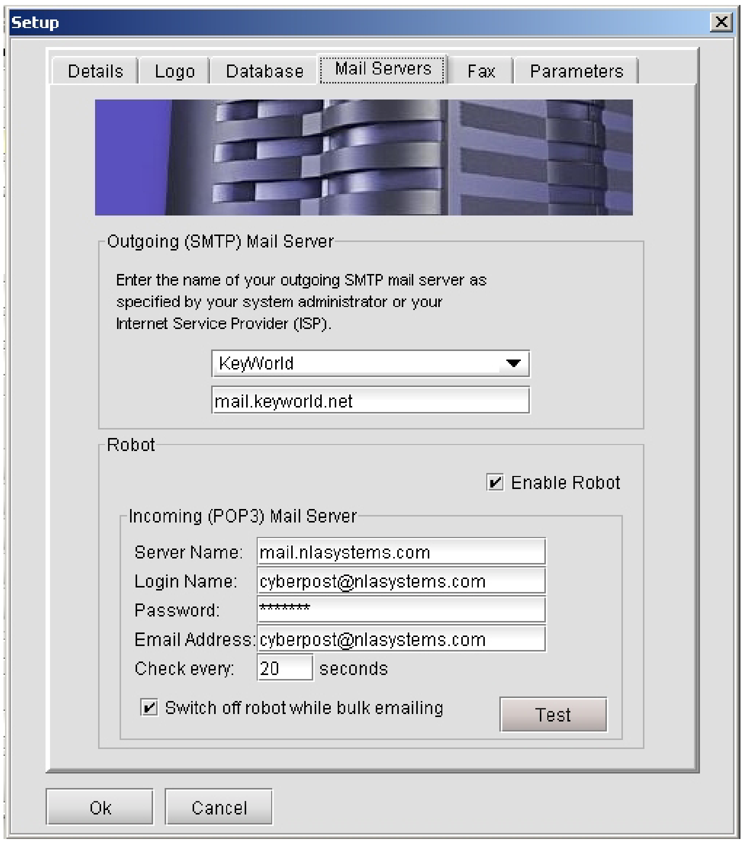
Database

An ODBC data source is a database that is registered with the ODBC driver. In Java we can use either the JDBC to ODBC bridge, or JDBC and a vendor-specific bridge to connect to the data source. To do this we need to specify the driver class and driver URL. Selecting a suitable driver from the "Driver Templates" combo box brings up templates for these two fields. All we have to do is fill in the specific information regarding our database.

Note to the administrator:

The JDBC template information comes from an xml file (jdbcTemplates.xml) in the cyberPOST2 working directory. This file can be edited to include other driver templates.

After we have entered our database connection details including login name and password, we can test our newly defined connection by clicking the 'Test Connection' button.



Outgoing (SMTP) Mail Server

Enter the name of your outgoing SMTP mail server as specified by your system administrator or your Internet Service Provider (ISP).

KeyWorld

mail.keyworld.net

Robot

☑ Enable Robot

Incoming (POP3) Mail Server

Server Name:	mail.nlasystems.com
Login Name:	cyberpost@nlasystems.com
Password:	********
Email Address:	cyberpost@nlasystems.com
Check every:	20 seconds

☑ Switch off robot while bulk emailing

Mail Servers / Robot

Outgoing (SMTP) Mail Server

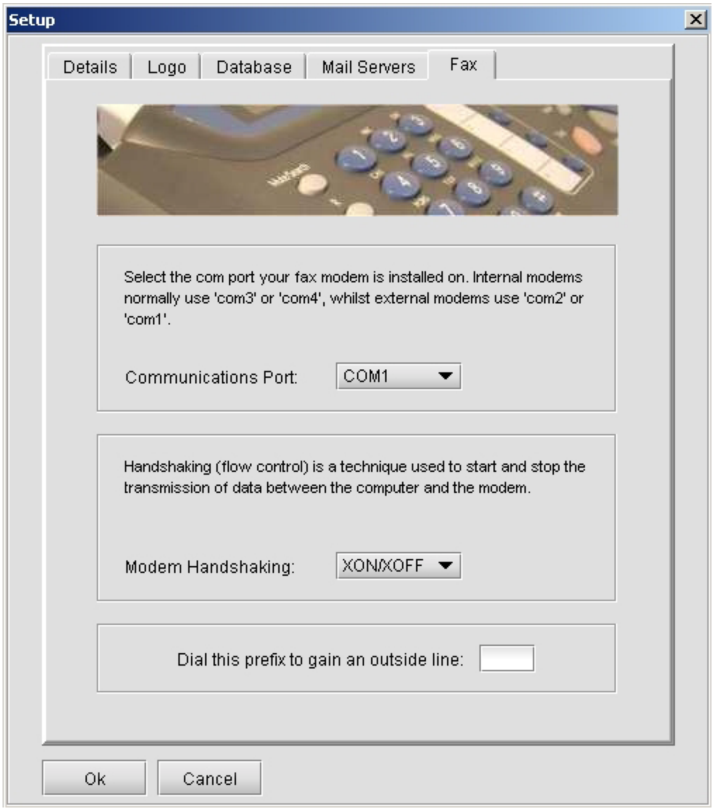
cyberPOST2 uses our mail server to independently send out our documents using SMTP (Simple Mail Transfer Protocol). Here we can select our service provider from the pull down menu list and the SMTP server will be filled in automatically. If our provider is not in the list, we can select 'Other' and type in the mail server.

Robot

To switch on the cyberPOST robot we check the 'Enable Robot' checkbox, and then enter the name of our incoming POP3 mail server. We also enter the mailbox's user name and password. To check that we entered the correct details we click the 'Test' button.

We should have a mailbox specifically for this function. Once the robot is enabled, it will regularly check the inbox for incoming requests for documents every number of seconds specified by us in the 'Check every' field. The robot will process all valid requests but will delete all the other messages. This is why this email address should be used exclusively for this purpose. In case of a multi-company installation, we can have a single mailbox for all companies. The robot will route the messages to their respective companies.

We can instruct the system to switch off the robot while bulk outgoing email is in process. We might want to do this if the robot is set to check for incoming requests at frequent intervals while the mail server is not local and as a result this would slow down outgoing mail. The robot is automatically switched on as soon as the process is finished. We can make use of this feature by checking the 'Switch off robot while bulk emailing' checkbox.



Select the com port your fax modem is installed on. Internal modems normally use 'com3' or 'com4', whilst external modems use 'com2' or 'com1'.

Communications Port: COM1

Handshaking (flow control) is a technique used to start and stop the transmission of data between the computer and the modem.

Modem Handshaking: XON/XOFF

Dial this prefix to gain an outside line:

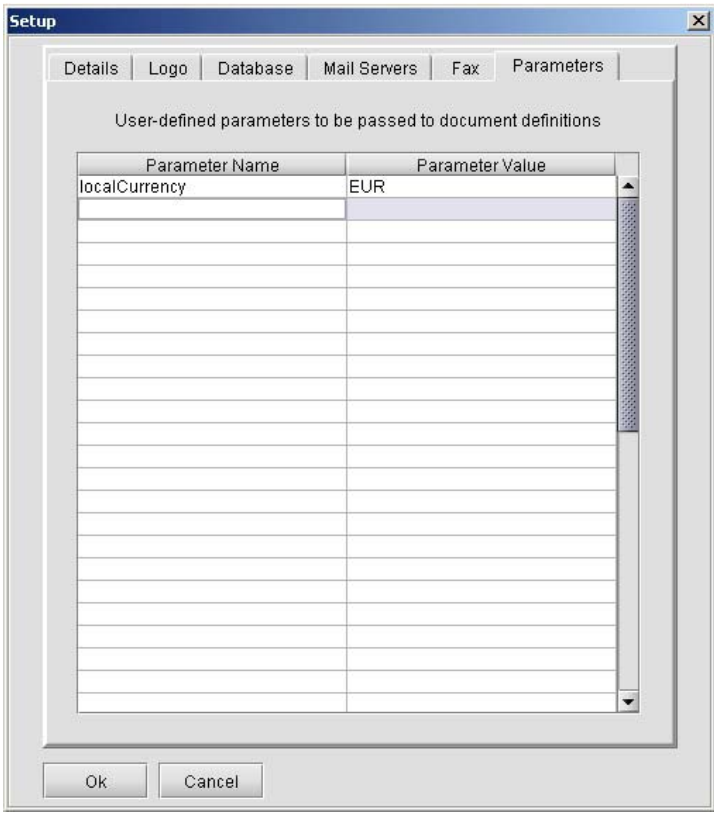
Fax

The fax process needs a fax modem installed on the local computer. This is where we specify its COM port number. To find this out from Windows we go to Start --> Settings --> Control Panel and then choose the 'Phone and Modem Options' icon. Internal modems are commonly configured as COM3 or COM4 while external modems are very often installed as COM1 or COM2.

Handshaking (flow control) is a technique used to start and stop the transmission of data between the computer and the modem. If we do not know what type of handshaking our modem uses and experience problems, then one thing we can try is to switch to XON/XOFF since RTS/CTS handshaking is set by default.

If our fax modem is connected to a PABX and we need to dial a prefix to get an outside line, we should enter this prefix in the field provided and make sure we add a comma as a suffix. This signals our modem to pause for a second or so.

Although this fax function can work under Linux it has not been tested yet at the time of writing this manual.



User-defined parameters to be passed to document definitions

Parameter Name	Parameter Value
localCurrency	EUR

User Defined Parameters

This is were we can declare any user-defined parameters. These will be passed on to the document definitions along with the other parameters. For example if we want to have our statements application to handle foreign currency we must pass a parameter that specifies our local currency to the report definition. In this way the report can distinguish between a local and a foreign currency account. To enter a new parameter we simply type in its name and its value in any one of the lines available. To remove it, we just have to blank the two fields in the line where it has been entered.

3

The cpxml Expressions

Throughout a cpxml file you can use expressions. Expressions can be a combination of literal characters and variables. The following are the types of variables we can use: -

Fields are represented by the field name wrapped in **$F{}.** Fields can be fixed-name fields corresponding to the unique ID tags in the cyberPOST2 user-interface or newly created ones, binded to data items and/or expressions. For example the field 'email' is represented by $F{email}.

XPATH expressions can be used and are wrapped in **$X{}.** For example the node value or content of the 'documentName' element can be represented by $X{/*/properties/documentName/text()}.

New to XPATH?
XPath is a language for finding information in an XML document. More details about XPATH can be found here http://www.w3schools.com/xpath/default.asp

Combo Displayed Items are the visual part of the combo box items. A combo field item consists of two values. The first value is the one actually displayed while the other is the auxiliary value that will actually be used when the item is selected. We refer to this value by the field name wrapped in **$D{}**. For example we should refer to the value of the currently selected item in the combo box 'combo0' as $F{combo0} and its currently selected display value as $D{combo0}.

System variables are wrapped in **$V{}**. System variable names are case-sensitive and should be in CAPS. For example today's date is represented by $V{TODAY}.

User-defined parameters described earlier can also be referenced by using the $V{} wrapper. The local currency user-defined parameter in our example would be referenced by $V{localCurrency}.

Invoked method returned variables are referenced by using the $I{} wrapper. These variables and their usefulness are described in the section titled 'Custom Method Invocation' in the next chapter.

All these variables are automatically passed as parameters to our report definitions. In the case of a Jasper report definition, all these variables are automatically packed in the java.util.HashMap that is expected by the Jasper engine.

Following is a list of system variables:-

Variable Name	Return Value
TODAY	System date in SQL YYYY-MM-DD format
YESTERDAY	$V{TODAY} - 1 day in SQL YYYY-MM-DD format
TOMORROW	$V{TODAY} + 1 day in SQL YYYY-MM-DD format
THISMONTHFROM	This month's 1st day - YYYY-MM-DD
THISMONTHTO	This month's last day - YYYY-MM-DD
COMPSERIALNUMBER	Licensed serial number
COMPNAME	Licensed company name
COMPADD1	User-defined company address lines 1-5 as inputted in Setup & Options - *Details*
COMPADD2	
COMPADD3	
COMPADD4	
COMPADD5	
COMPVATNO	User-defined Company VAT number as inputted in Setup & Options – *Details*
COMPTEL	User-defined company telephone numbers as inputted in Setup & Options - *Details*
COMPFAX	User-defined company fax number as inputted in Setup & Options – *Details*
COMPEMAIL	User-defined company contact email address as inputted in Setup & Options – *Details*
COMPWEBADD	User-defined company web-site URL as inputted in Setup & Options - *Details*

COMPPOP3MAILADDRESS	The email address that is used by the robot to receive requests.
COMPCOLOURLOGO	The filename together with its full path of the company's logo.
COMPMONOLOGO	The filename together with its full path of the company's logo in black & white. This will be used in faxed documents
WORKINGDIRECTORY	working directory (Windows: c:\cyberPOST2, Linux: <user-home>/cyberPOST2)
TOSENDBY	Current document destination represented by a single character. E – Email / F – Fax / P – Print / V - Preview S – SMS (for future use)

4

Building the cpxml interface

Following is a description of all the XML elements that builds a valid cpXML interface file. After each section's description, a DTD excerpt pertaining to that section is included.

New to DTD?
The purpose of a DTD (Document Type Definition) is to define the legal building blocks of an XML document. It defines the document structure with a list of legal elements. You can learn more about DTD here http://www.w3schools.com/dtd/default.asp

The root element should be 'cyberPOST2'. It splits up into 6 sections and we shall go through them one by one. Our document distribution application may not necessarily use all of these sections.

XML syntax (DTD):

```
<!ELEMENT cyberPOST2 (properties, sql?, field*, reportParameters?, method?, admin?)>
```

Properties

The <**properties**> element can have two attributes **batchNoPrompt** and **robotOnly**. Setting **batchNoPrompt** to "true" enables the batch process to start sending out messages without prompting with the number of messages to send and without waiting for a confirmation. Setting **robotOnly** to true enables the document definition to be excluded from the Document Combo Box thus preventing the user from sending out this document. However this will be available to the robot and will be sent out if requested by a robot request. The absence of these attributes is equivalent to setting them "false"

The <**properties**> element includes the following elements that define how cyberPOST2 behaves when the particular document definition is switched to.

<documentName> is the name we assign to our document and it is the name that will show in the GUI '**Document'** selection combo box.

Under the <**emailSettings>** element, we define what appears in the subject and body of our outbound email that delivers our dynamic document.

Note: cyberPOST2 comes with some sample applications installed. This tutorial uses them to provide examples. It is advisable to print out the example cpxml files so that you can better follow the references made. These can be found in the sub-folder 'reports' of the working directory.

The <**subject>** element can contain an expression that describes the email subject. For example, in our debtor statements application we want to show the document name (Statement) and the account number. Assuming we have binded the field 'key1' to the account number column in our data source, the expression to achieve this would be "Statement - $F{key1}".

Your company name will automatically be prefixed to the email subject. Assuming our company name is "NLA Systems Ltd.", the recipient of our email will actually see "NLA Systems Ltd. – Statement-E01" in his subject, also assuming that his account number is "E01".

The <**body>** element is defined in the same way as the subject. However if this definition turns out to be large and clutters our file, we can write the element content to a separate file and provide its full path and file name instead. If a filename is assigned without its full path, it is assumed that it is to be found in the "reports" folder in the working directory. Therefore the value of the <**body>** element can be either actual code or a filename pointing to the file containing the code. We have to specify this through the **valueType** attribute, which should be one of two values "filename" or "actualCode". Default is "actualCode". The body content can be either text or HTML and the attribute **type** specifies this. Valid values for this attribute are "text**"** or "html**"**. Default is "text".

The **<report>** element **type** attribute specifies which report writer we used to design the document template for our particular document application. This must have one of the following values "Jasper", "Crystal", or "None". If a report writer was used to create a report template we can use the **name** attribute to identify the template file. If such a name is not specified cyebrPOST2 assumes that the report template bears the same name as the cpxml file but with the appropriate extension. For example in our statements examples we have 'nlaStatement.cpxml' as our cpxml filename and 'nlaStatement.jasper' as our Jasper Reports template. In this case the attribute **name** can be done without.

Default **type** is "None" and this means that your document application does not use a report writer but will only use the body expression as a document.

An example of a text body definition is as follows:-

<**body**>Dear $F{contactDesig} $F{contactName} $F{contactSurname}

We wish to draw your attention to your account number $F{key1} which has an overdue balance of $F{balance} for quite a while. Kindly settle all overdue amount at your earliest convenience.<**/body**>

JDBC data source

The **<dataSource>** element **type** attribute specifies whether cyberPOST2 is to retrieve its data from a JDBC/ODBC connection or parse its fields from a plain text file. The two valid values for this attribute are "jdbc" and "text". Default is "jdbc"

Parsed text data source

If our data source is a text file that contains a stream of documents, we need to specify the filename (with its full path) of this text file and a string of characters that mark the beginning and end of each document. This is useful information for the text parser to know where a document starts and where it ends. This information is specified in the 3 elements **<filename>**, **<startDocumentTag>** and **<endDocumentTag>** respectively. The actual parsing of the fields is explained in the chapter "Field section".

cyberPOST2 is multi-company application. If we are a group of companies, we can install just one application that handles all companies in the group. In this case cyberPOST2 will prompt us with a selection list of all our companies every time it is launched. All Document Definitions that we have defined will be made available to all companies by default.

If we do not want the Document Definition we are currently building to show in one or more of our companies, we just enter the serial number of each company that we want to exclude in a separate **<companySerNo>** element under the **<excludeInCompany>** element. Any number of **<companySerNo>** elements can reside within **<excludeInCompany>.**

All cyberPOST2 activities are logged into a log file including an entry for each dispatched document with its date and time. The **<logEntry>** element defines what will show in each log entry of a document dispatch event. For example to show the document name, document number and email address in our debtor statements application we should have the value "Statement - $F{name} - $F{key1} - $F{email}" in our **<logEntry>** element. There is no need to add a date variable, since each entry is automatically time stamped.

Properties section XML syntax (DTD):

```
<!ELEMENT properties
        (documentName,emailSettings,report,dataSource?
        ,excludeInCompany?,logEntry?)>
  <!ATTLIST properties      batchNoPrompt (true|false) "false">
  <!ATTLIST properties      robotOnly (true|false) "false"
<!ELEMENT documentName (#PCDATA)>
  <!ELEMENT emailSettings (subject, body?)>
    <!ELEMENT subject (#PCDATA)>
    <!ELEMENT body (#PCDATA)>
      <!ATTLIST body type (html|text) "text">
      <!ATTLIST body valueType (filename|actualCode) "actualCode">
  <!ELEMENT report EMPTY>
  <!ATTLIST report type (Jasper|Crystal|None) "None">
  <!ATTLIST report name NMTOKEN #IMPLIED>
  <!ELEMENT dataSource
        (filename,startDocumnentTag,endDocumentTag?,bindData?)>
    <!ELEMENT filename (#PCDATA)>
    <!ELEMENT startDocumnentTag (#PCDATA)>
    <!ELEMENT endDocumentTag (#PCDATA)>
  <!ATTLIST dataSource type (jdbc|text) "jdbc">
  <!ELEMENT excludeInCompany (companySerNo*)>
    <!ELEMENT companySerNo (#PCDATA)>
  <!ELEMENT logEntry (#PCDATA)>
```

SQL Section

If our data source is a JDBC connection, we need to provide cyberPOST2 with the proper SQL statements to execute in the various stages of its processes. These are defined within the <**sql**> element which is sub-divided into <**master>**, <**transaction>**, <**whereClauses>** and <**orderByClauses**>. These sub-elements are designed to contain SQL fragments or clauses with which cyberPOST2 can build the required statements to retrieve the correct information.

New to SQL?
SQL is a standard computer language for accessing and manipulating databases. You can get all the basics here http://www.w3schools.com/sql/default.asp

In <**master>**, we can provide the FROM, WHERE and ORDER BY clauses to retrieve the necessary columns from the master table. Normally, the master table contains details of the document recipients. For example in our debtor statements application, we have the debtors' master table name in the <**masterFrom**> element. In our demo data this is 'stkcst'. The content of <**masterFrom**> is not limited to just a table name but any SQL syntax you might have in any SQL FROM clause. For example in a case where master details are in separate tables, we can specify all the table names separated by commas or we might want to JOIN our tables. In this case we would need to define the relationship or link between these tables. For example if our master details exist in two tables "cstDetails" and "cstAddresses" and they are linked by their key "cstCode" we would define our <**master**> element this way:-

```
<master>
   <masterFrom>cstDetails, cstAddresses</masterFrom>
   <masterWhere>cstDetails.cstCode = cstAddresses.cstCode</masterWhere>
   <masterOrderBy></masterOrderBy>
</master>
```

All the cpxml expressions described earlier can be used anywhere in these clauses. For example, if we append the line below to our example <**masterWhere**> content above, cyberPOST2 will evaluate this expression immediately before execution.

```
 AND cstDetails.cstTel like '$F{extra1}%'
```

What would happen here is that whatever the user enters in the text field **extra1** replaces the characters $F{extra1} in our WHERE clause. Specifically, if the user keys in "356" in his extra1 field, the expression evaluates to

```
AND cstDetails.cstTel like '356%'
```

This way, we have created a built-in filter that would select for us only those customers whose telephone number starts with a certain number of characters.

At this point one may ask; What about the SELECT clause? Well this is automatically taken care of by the application. A list of needed columns is extracted from our <**field**> sections. We will come to this soon.

When we have a master/transaction scenario we need to define the <**transaction**> element too. We have this as an example in our Invoicing sample application (nlaInvoice.cpxml). The <**transaction**> element is sub-divided into <**transactionFrom>**, <**transactionWhere>** and <**transactionOrderBy>** and they are defined in the same manner as in the <**master**> element. In most cases the <**transactionWhere**> element contains a link or relationship to the master table/s. In our Invoicing sample application, "bisinv.customer = stkcst.code" achieves this.

cyberPOST2 can distribute documents:-

- individually by interactively choosing the desired recipients
- by batch using user-defined filters to specify and control list of recipients
- as a response to a direct client request to the built-in robot, Zigineeze

In order to enable cyberPOST2 retrieve the required information to execute each of these processes, we need to put in their respective WHERE clause within the <**whereClauses**> element.

To pick recipients for one-by-one delivery, one can type-in the record's key in its text field and then press <ENTER> or else he can browse through the records. In the case of our debtor statements example, the user types-in the account number of the particular debtor he wants to send his statement to, and presses <ENTER>. Alternatively, he/she can double-click the account number field to open the browse dialog. For both situations the system needs to trigger a "Find Master Record" method that uses the contents of both the <**master**> and the <**findMasterWhere**> elements to build its SQL statement. Master/Transactions applications like our Invoicing example need to have <**findTransactionWhere**> defined for the same reason.

When sending documents by batch, unless otherwise specified by the user-defined filters, the system will send the currently selected document to all the recipients in the master table. By means of the <**batchSendWhere**> element we can install a permanent filter that constraints the recipient list in the batch process.

The cyberPOST2 document dispatch robot checks whether the requester is authorized to receive the document in question and immediately compiles and sends out this document. If we want to install other constraints to this process the <**robotWhere**> element is the place to specify this.

Note: *For security reasons, each email request is automatically checked by the robot to ensure that the sender email address corresponds to the email address in the correct record in our database and that the document he or she is asking for pertains to him or her. This way, nobody can receive some other recipient's document.*

We can enable any field in the user interface to be browsed. We may dictate this functionality in the <**field**> section. When a field is enabled for browsing, it is colored light yellow. When the user double-clicks this yellow field, the browse dialog containing the relevant records pops up. What columns show in these dialogs is also defined in the <**field**> section but which records come up are dictated by the WHERE clause in the <**browseWhere**> element. The way to specify which WHERE clause pertains to which field is by stating its field name in the **field** attribute. In our statements example, we have enabled browsing for the account number field and we have defined some search conditions. We need to be able to type-in part of the debtor's name in the name field and have cyberPOST2 retrieve all names that match this input. We have achieved this through the following SQL fragment.

<**browseWhere** field="key1">stkcst.cstname like
'%$F{name}%'</**browseWhere**>

Sorting the queries for the batch and the browse processes involves inserting ORDER BY clauses in the <**batchSendOrderBy**> and the <**browseOrderBy**> elements respectively. The <**browseOrderBy**> element needs to have the attribute **field** assigned to its relative field name in the same way that is done for <**browseWhere**>.

A useful debugging feature is the ability to output all the SQL statements constructed by cyberPOST2 to the standard output. We can instruct the system to do this for us by assigning the <**sql**> attribute **verbose** to "true".

SQL section XML syntax:

```
<!ELEMENT sql (master, transaction, whereClauses, orderByClauses)>
<!ATTLIST sql verbose (true|false) "false">
  <!ELEMENT master (masterFrom, masterWhere?, masterOrderBy?)>
    <!ELEMENT masterFrom (#PCDATA)>
    <!ELEMENT masterWhere (#PCDATA)>
    <!ELEMENT masterOrderBy (#PCDATA)>
  <!ELEMENT transaction
       (transactionFrom, transactionWhere?, transactionOrderBy?)>
    <!ELEMENT transactionFrom (#PCDATA)>
    <!ELEMENT transactionWhere (#PCDATA)>
    <!ELEMENT transactionOrderBy (#PCDATA)>
  <!ELEMENT whereClauses
       (findMasterWhere, findTransactionWhere?,
       batchSendWhere?, robotWhere?, browseWhere*)>
    <!ELEMENT findMasterWhere (#PCDATA)>
    <!ELEMENT findTransactionWhere (#PCDATA)>
    <!ELEMENT batchSendWhere (#PCDATA)>
    <!ELEMENT robotWhere (#PCDATA)>
    <!ELEMENT browseWhere (#PCDATA)>
    <!ATTLIST browseWhere field NMTOKEN #REQUIRED>
  <!ELEMENT orderByClauses (batchSendOrderBy, browseOrderBy*)>
    <!ELEMENT batchSendOrderBy (#PCDATA)>
    <!ELEMENT browseOrderBy (#PCDATA)>
    <!ATTLIST browseOrderBy     field NMTOKEN #REQUIRED>
```

Field Section

This section of our cpxml is where we declare our list of fields or variables that we can then use in our expressions. In a <**field**> element we define where the field gets its value from, and other user-interface features. Each field has to have a unique name that is assigned through the **id** attribute. You can choose any name for your fields except the reserved names that are assigned to a component in the user interface. Each component's assigned field name is shown in Figure 3 as its field tag. To give an example, if we want to get input from the first text field tagged "key1" we should have <**field** id="key1" …

In the case of these gui-binded fields, we can set their label tags, tooltip text and a default value through the <**label**>, <**toolTipText**> and <**default**> elements respectively.

Fields can be binded to JDBC table columns or data parsed from a text file.

JDBC data-binded fields

If we want to bind a field to data coming from a JDBC connection, all we have to do is to assign a table column name in the <**bindData**> element. This should be in the form of table.column so that cyberPOST2 can distinguish between columns coming from master or transaction tables. Here is an example of how we binded the debtor's telephone number to the GUI text field component "extra1" in our sample statements application.

```
<field id="extra1">
   <label>Tel</label>
   <bindData>stkcst.tel</bindData>
</field>
```

A user interface component can be binded to more than one data item and can have literal text parts. This is done through the <**bindDataGroup**> element within which we can have a combination of <**bindData**>, <**text**> and <**newline**> elements. A useful example for this is illustrated in our sample applications. In our demo database we defined the customer address as three columns add1, add2 and add3 rather than a single text field. This is how we binded these values separated by a new line to our GUI text field tagged "text" (Figure 3): -

```
<field id="text">
   <label>Address</label>
   <bindDataGroup>
      <bindData>stkcst.add1</bindData>
      <newline/>
      <bindData>stkcst.add2</bindData>
```

```
    <newline/>
    <bindData>stkcst.add3</bindData>
  </bindDataGroup>
</field>
```

Other examples of where this might come useful are: -

- Cases where one can have designation, name and surname as three columns formatted in a single GUI text field such as “name”. The following XML achieves this.

```
<field id="name">
  <label>Name</label>
  <bindDataGroup>
    <bindData>stkcst.desig</bindData>
    <text> </text>
    <bindData>stkcst.name</bindData>
    <text> </text>
    <bindData>stkcst.surname</bindData>
  </bindDataGroup>
</field>
```

- Telephone number with separate country codes and extension numbers formatted and separated by special characters is defined in the following manner.

```
<field id="extra1">
  <label>Tel</label>
  <bindDataGroup>
    <text>)+(</text>
    <bindData>stkcst.prefix</bindData>
    <text>) </text>
    <bindData>stkcst.tel</bindData>
    <text> - Ext: </text>
    <bindData>stkcst.extension</bindData>
  </bindDataGroup>
</field>
```

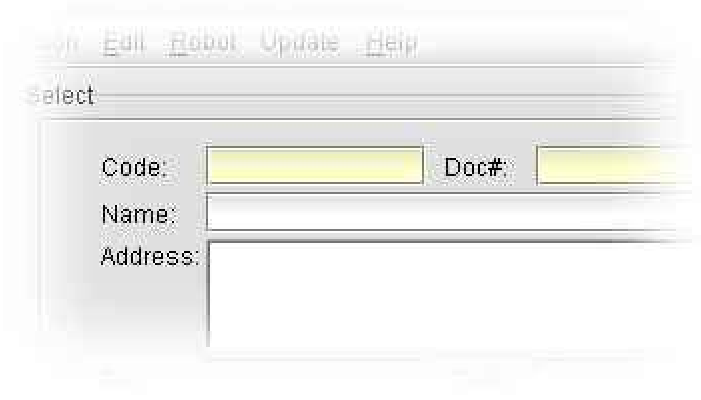
If our field is binded to a GUI combo box, we need to define the list of line items that it will contain. Each item consists of a display value and return value. These are assigned through the <**display**> and <**value**> elements within <**comboItem**>. Let us use our sample invoicing application as an example. The invoice archiving table in our demo database houses not just invoices but credit notes, cash sales and cash sale returns. The row layout is the same for all documents and the rows are distinguished through the column trans_type. Therefore, to request one of the documents we need both its document number and its transaction type inputted by the user to be passed on to the report definition. The trans_type column consists of a single character and is one of four values; A, B, C or D. The combo box tagged "combo0" has been defined as follows.

```
<field id="combo0">
  <label>Sub-Document</label>
  <toolTipText>Choose between one of these sub-documents</toolTipText>
  <comboItem>
    <display>Invoice</display>
    <value>A</value>
  </comboItem>
  <comboItem>
    <display>Cash Sale</display>
    <value>B</value>
  </comboItem>
  <comboItem>
    <display>Credit Note</display>
    <value>C</value>
  </comboItem>
  <comboItem>
    <display>Cash Return</display>
    <value>D</value>
  </comboItem>
</field>
```

Supposing the third item "Credit Note" is selected, the field representations $F{combo0} and $D{combo0} contain "C" and "Credit Note" respectively.

A combo box can have any one of its items set as default through the <**default**> element. Here, the display value should be given.

A GUI text field can be enabled for browsing by simply inserting a <**browseTable**> element within <**field**>. This consists of a list of <**column**> elements that each contains a table column name whose data will show in that particular browse column. We must also make sure to have the proper SQL defined for this field as described in the SQL section earlier. A field can also be enabled to open the file browser or the date browser. This is done through the type attribute by setting it to "fileBrowser" or "dateBrowser" respectively.



When a field is browse enabled, its color changes to light yellow. It can be double-clicked to open the browse table. The tooltip text is appended with the words "DOUBLE-CLICK to browse."

Fields can be binded to JDBC table columns or data parsed from a text file.

Parsed text data-binded fields

If our data source is a plain text file and is a series of formatted documents, we can assign specific regular areas in this text to be binded to a cpxml field through the <**parseText**> element. There are two ways available to do this.

The first one is by specifying a particular line and particular character position to start reading the field from and specifying its length. Both line and character positions start from 1. If the field length specified is 0, the field will be read until the end of the line. We can set these three values within a <**byPosition**> element, in <**lineNo**>, <**charPos**> and <**length**> sub-elements.

The second option is the tagging option and comes useful when the sequential documents are not uniformly formatted. Suppose a field is sometimes on line 2 and other times on line 3, depending on certain conditions. Our line/position/length method will fail, so instead, we should use the tagging option. This is achieved by assigning a <**startTag**> and an <**endTag**> element within <**byTags**>. These tags are a string of characters that can always be found just before and just after the data field in the text file. The parser will read the characters between these tags (exclusive of the tags themselves).

If <**endTag**> is not assigned, the field will be read until the end of the line. The disadvantage of this method is that it is slower than the absolute position method and we normally resort to it only if our input text document is not uniform.

Field section XML syntax:

```
<!ELEMENT field (label?, bindData?, toolTipText?,browseTable?,
                 bindDataGroup?, comboItem*, parseText?)>
<!ATTLIST field          id ID #REQUIRED
                         required (true|false) "false"
                         displayOnly (true|false) “false”
                         trim (true|false) "false">
   <!ELEMENT label (#PCDATA)>
   <!ELEMENT toolTipText (#PCDATA)>
   <!ELEMENT default (#PCDATA)>
   <!ELEMENT bindData (#PCDATA)>
   <!ELEMENT bindDataGroup (bindData|text|newline)*>
      <!ELEMENT text (#PCDATA)>
      <!ELEMENT newline EMPTY>

   <!ELEMENT browseTable (column*)>
   <!ATTLIST browseTable type (fileBrowser|dateBrowser)
#IMPLIED >
   <!ELEMENT column (#PCDATA)>
   <!ELEMENT comboItem (display, value)>
      <!ELEMENT display (#PCDATA)>
      <!ELEMENT value (#PCDATA)>

   <!ELEMENT parseText (byPosition?, byTags?)>
      <!ELEMENT byPosition (lineNo, charPos, length)>
         <!ELEMENT lineNo (#PCDATA)>
         <!ELEMENT charPos (#PCDATA)>
         <!ELEMENT length (#PCDATA)>
      <!ELEMENT byTags (startTag, endTag)>
         <!ELEMENT startTag (#PCDATA)>
         <!ELEMENT endTag (#PCDATA)>
```

Report parameters

If cyberPOST2 makes use of a report writer such as Jasper Reports to create its dynamic documents, in nearly all cases we need to pass parameters to it. This is done by inserting a <**parameter**> element within the <**reportParameters**> section for each of the parameters that the report writer is expecting. As in most places throughout a cpxml, the use of cpxml expressions is allowed here as well. The element <**parameter**> has two attributes. The attribute **name** should be assigned with the parameter name as it was declared in the report definition. The other attribute **blankOnAnyBlankField** is an optional *boolean* attribute and can be used if a <**parameter**> expression contains one or more cpxml fields. If this attribute is set to "true" and the value of any one of the fields is null or blank, then a blank parameter is passed on to the report. Its use may not be obvious immediately. To give an example of where this came useful we once again refer to our statements example: -

<**parameter** name="fromDate" blankOnAnyBlankField="true"><![CDATA[and stkdbtr.trdate>='$F{field1}']]></**parameter**>

Here we are passing an SQL fragment as a parameter to the Jasper Report we used to define our statement document. If the user enters a date in the GUI text field "field1", then we need to pass the whole evaluated expression to the report. On the other hand, if "field1" is blank, we need to pass a blank parameter to the report so that no filtering by date occurs. In other words it is an all-or-nothing instruction.

Report Parameters XML syntax:

```
<!ELEMENT reportParameters (parameter*)>
   <!ELEMENT parameter (#PCDATA)>
   <!ATTLIST parameter name NMTOKEN #REQUIRED>
   <!ATTLIST parameter
                          blankOnAnyBlankField (true|false) "false" >
```

Custom method invocation

Apart from dealing with requests for documents, cyberPOST2 can be made to trigger custom external Java methods that the administrator may create. These Java methods should be able to make their results available to cyberPOST2 to be used in cpxml expressions. Therefore, these methods should obey the following simple rules in order for it to properly interface with cyberPOST2 and/or the robot.

- Their constructor should have a java.util.HashMap argument containing any parameters that we need to pass to the custom java method.
- They should return a java.util.HashMap containing any return values we wish to make available to be processed. The content of these return values are referenced in cpxml expressions by wrapping their name in $I{}.
- The compiled Java classes should be located in the 'cpCustom' sub-folder in the cyberPOST2 working directory.

One of the objects in the HashMap argument is always passed by cyberPOST2 and is named 'dbConnection'. It is of type java.sql.Connection. This object enables us to make use of the current application database connection instance in our custom Java class.

If the custom method is intended to be invoked through the robot, the body of the document request email should contain valid XML code as described in the next chapter. It can contain arguments to be passed on to the custom method.

To set up the custom method in our cpxml file, the method name should be declared with its full class path in the element <**name**> within the <**method**> element and is given a unique tag through the **id** attribute. This id is then used by cyberPOST2 and in the XML body of the email request to identify this method.

After our custom method completes execution and returns its java.util.HashMap containing its return values, we can send to the recipient or the sender of the email request to the robot any of these values or results and also a result message depending on the value of any of these returned values. This result message may also contain cpxml expressions. As stated above, each return value can be represented by wrapping its name in $I{} and used in any cpxml expression.

To define a result message we insert a <**result**> element within the <**method**> element. We give this result message an id through the **id** attribute. This message is represented in a cpxml expression as this id but wrapped in $R{}.Now we have to specify a message for each of all the possible values of a return variable that we wish to report to the recipient. Each of these messages is entered as text in an <**onResult**> element. Its attribute name is the return variable name as declared in the invoked Java method or class and the value of this attribute is the return value of this variable in relation to the required message text.

We can illustrate all this in the example below taken from a real life application that we implemented for the Xerox representative in Malta namely Image Systems Ltd. They rent out and also lease out Xerox copiers and charge their clients by the number of copies made. As a result, they go through the laborious task of collecting all the meter readings from each machine every month. They run their batch billing process every month. Collection of meter readings is normally done through the mail, by means of phone calls and by visiting technical personnel. Recently they have had a number of clients submitting their meter readings through ISL's web site.

Simply put, our cyberPOST2 application was configured to send out emails to all clients that have rented and leased machines. Each email contains a link that takes each client to his online meter submission form. As soon as the ISL client submits his/her meter reading, an email to the robot is sent. The robot parses the xml content of this email to retrieve the method id to invoke together with the necessary arguments. A java class **invoke.Xermeter.saveMeterReading** was written to effect the necessary validations on the meter reading and then post this reading if it is valid. This method was given an id "postMeterReading". In this method, a return String variable **result_code** is declared and can contain "systemError", "smaller" or "OK" as soon as execution of the method is finished. If a java error occurs, the value "systemError" is returned in **result_code**. A meter reading cannot be smaller than the last submitted reading and this check is effected at source in the input form, but since the robot can receive the same submission email from another source such as a simple email client, this check is also done in the "postMeterReading" method. If this reading turns out to be smaller than the last submitted one, the value "smaller" is returned in **result_code** and the reading does not get posted but if all goes well and a valid meter reading gets posted the value "OK" is returned. As you can see from the cpxml excerpt below three messages were defined one for each circumstance. This result message is represented by the cpxml expression $R{INVOKE_RESULT} which is the result id wrapped in $R{}.

```
<method id = "postMeterReading">
   <name>invoke.Xermeter.saveMeterReading</name>
   <result id = "INVOKE_RESULT">
      <onReturn result_code = "systemError"><SYSTEM ERROR: Meter
                                 reading posting procedure generated an  error
                                  while posting your entry. We will look into it and
                                  contact you shortly. Sorry for the inconvenience
                                 </onReturn>
      <onReturn result_code = "smaller">USER ERROR: Meter reading
                                  submitted is smaller than the previous one
                                  ($I{lastReading} on I{lastReadingDate})..Kindly
                                  re-submit correct reading.</onReturn>
      <onReturn result_code = "OK">Your posting has been successfully
                                  posted. (Machine Reference: $I{machine_ref}
                                  Meter Reading: $I{reading})</onReturn>
   </result>
</method>
```

Another example is an implementation for a local distributor MGS Ltd based in Iklin, Malta.

The problem MGS needed to find an efficient and cost effective way for their traveling sales reps to transmit their customer orders as soon as they conclude them. In this way action on delivery is taken immediately. A PDA will have to be used but to connect this to their back end system through terminal services via GPRS would be both an expensive and a slow proposition, at least with our present cell phone offers and infrastructure.

The solution Our PDA ordering system was designed to input the customer order and email it in xml format to the cyberPOST2 robot which then invokes a specially written Java method to post this order in the orders database and to print a copy of this order to the warehouse printer. Below is an excerpt of this implementation's cpxml file, particularly the part that defines the method invocation.

```
<method id="postPdaOrder">
   <name>invoke.PdaSales.postSalesOrder</name>
   <result id='ORDER_RESULT'>
      <onReturn RESULT_CODE="systemError">Sales Order posting
                 returned an error while posting your entry.
                 - $I[JAVA_ERROR}</onReturn>
      <onReturn RESULT_CODE="">Your posting has been
                 successfully received and posted.</onReturn>
   </result>
</method>
```

Custom method invocation XML syntax:

```
<!ELEMENT method (name, result*)>
<!ATTLIST method    id NMTOKEN #REQUIRED>
   <!ELEMENT name (#PCDATA)>
   <!ELEMENT result (onReturn*)>
      <!ATTLIST result         id NMTOKEN #REQUIRED>
      <!ELEMENT onReturn (#PCDATA)>
         <!ATTLIST onReturn  * NMTOKEN #IMPLIED>
```

Note: The last line of the DTD above is not valid DTD code but attempts to highlight the fact that <**onReturn**> attributes can bear any name. This is because this name is derived from the invoked method's returned variable names. It is not possible to represent this using DTD.

Admin Section

As explained earlier the built-in document delivery robot verifies the requester's sender email address before sending out the requested document. However a system administrator or some other authorized person might want to request any document. This can be done by including his or her email address in an <**email**> element within the <**admin**> section. You can have any number of <**email**> elements.

Admin XML syntax:

```
<!ELEMENT admin (email*)>

   <!ELEMENT email (#PCDATA)>
```

Zigineeze. The dynamic document robot.

Zigineeze is a background process in cyberPOST2 that can be switched on and off. When it is on, upon a user-defined interval of time, it connects to a dedicated POP3 mailbox that is also user-defined, and downloads all messages for processing. It does not matter what the email subject is since this is ignored. Each of these emails should contain valid Zigineeze XML. If it does not, then it is discarded and while a log entry is posted, the sender is notified that his request was invalid and has not been processed. Otherwise, it then extracts the sender-email-address from this message and checks it against the database and if it exists, checks whether the document he or she is requesting belongs to his or her account. If the result of this check is negative a log entry is posted and the sender is notified of the refusal of his request. Otherwise, the document is duly filled in and sent to the requester via email by default, or by fax if specified in the request. The security of this system lies in its simplicity. As mentioned earlier, the robot has also the ability to invoke a custom Java method and also pass to it all the necessary parameters. Upon completion, it then informs the requester of the result. All the time, communication is done via email. The advantage of this method is that our customer can get real-time information from us 'on demand' but at the same time they do not have a direct connection to our data. The other two advantages of this system are the simplicity of deployment and its subsequent maintenance and its considerable low cost especially when compared to a fully blown application server based system. It goes without saying that cyberPOST2 is limited to a certain number of applications and is not here to replace the application server!

All the time, communication is done via email. The advantage of this method is that the customer can get real-time information 'on demand' but at the same time they do not have a direct connection to our data.

Zigineeze request XML

A request to Zigineeze for a particular document is effected through an email containing some XML instructions.

The root element of any Zigineeze request should be <**cyberPOSTRequest**>. Within this root element, the following elements are compulsory: -

<**companySerNo**> cyberPOST2 is a multi-company application and Zigineeze can handle all the document definitions of all the companies installed. Therefore a company serial number to point to the application database is required.

<**document**> Zigineeze needs to know which document type the requester needs. This should be the same name that was given to the document definition (the cpxml filename without the extension)

We can fill-in the values of any cpxml field by making use of the <**field**> element. These values will be used by cyberPOST2 to evaluate its cpxml expressions just as if input came interactively from the GUI, and if a report definition is used, all the necessary parameters are passed to it. The <**field**> element consists of two sub-elements <**id**>, which contains the name of the cpxml field and <**value**>, which contains the value to be set. In the case of a combo box field the value to set should be the display value since it is more meaningful to the requester.

If not specified, Zigineeze will reply to a document request by email. However the requester may specify another form of delivery through the <**replyBy**> element. This can contain "email", "fax" or "print". The last option will send the report to the system's default printer.

It is probably asking too much from the requester to type-in all this code to request a document so we must make it as easy as can be for him or her. In one's web-site, one should have a suitable submit form, most probably with some input validations. Upon the pressing of the submit button, an email with the described XML code can be sent to the Zigineeze email address.

Another way to facilitate the use of Zigineeze XML is to provide mailto hyperlinks wherever they are allowed. For example in our statement application, the PDF document generated by Jasper Reports contains a hyperlink for every transaction in the reference column. The recipient can click on a particular hyperlink if he/she wants to receive the relevant Invoice, Credit Note or Receipt. In this case the default email client opens up with the email to be sent. The user only needs to press the 'Send' button.

If an <**email**> element is included, the text in this element will be parsed to get the reply-to email address or the recipient's email address. If this element is not included Zigineeze will extract this email address from the "From" address of the email request.

Zigineeze XML Syntax

```
<!ELEMENT cyberPOSTRequest
                    (companySerNo, document, field*, replyBy?, invoke*)>
   <!ELEMENT companySerNo (#PCDATA)>
   <!ELEMENT document (#PCDATA)>
  <!ELEMENT email (#PCDATA)>
   <!ELEMENT field (id, value)>
      <!ELEMENT id (#PCDATA)>
      <!ELEMENT value (#PCDATA)>
  <!ELEMENT replyBy (#PCDATA)>
   <!ELEMENT invoke (method, argument*)>
      <!ELEMENT method (#PCDATA)>
      <!ELEMENT argument (#PCDATA)>
    <!ATTLIST argument name  ID #IMPLIED>
```

Zigineeze request example 1

This is an example of a Zigineese request for Invoice number 3306 from company with serial number 145.

```
<?xml version="1.0" encoding="UTF-8"?>
<cyberPOSTRequest>
  <companySerNo>145</companySerNo>
    <document>nlaInvoice</document>
    <field>
      <id>combo0</id>
      <value>Invoice</value> <!--Invoice/Cash Sale/Credit Note/Cash Return-->
  </field>
    <field>
      <id>key2</id>
      <value>3306</value>    <!-- Document Number -->
  </field>
  </cyberPOSTRequest>
```

Zigineeze request example 2

This is an example of a Zigineese request for a statement of account number "H01" containing all transactions between 1st January 2004 and 31st December of the same year. The document is to be sent by fax through cyberPOST2's faxing capability.

```
<?xml version="1.0" encoding="UTF-8"?>
<cyberPOSTRequest>
 <companySerNo>1</companySerNo>
  <document>nlaStatement</document>
  <field>
     <id>key1</id>
     <value>H01</value>          <!-- Account Number -->
  </field>
  <field>
    <id>combo1</id> <!--Statement Type: "Outstanding" / "Full Account"-->
    <value>Full Account</value>
  </field>
  <field>
    <id>field1</id>    <!-- Trsactions From Date (YYYY-MM-DD format) -->
    <value>2004-01-01</value>
  </field>
  <field>
     <id>field3</id>  <!-- Trsactions To Date (YYYY-MM-DD format) -->
     <value>2004-12-31</value>
  </field>
  <field>
      <id>field0</id>            <!—Message to show in document -->
      <value>cyberPOST robot response (Requested)</value>
   </field>
   <replyBy>fax</replyBy>    <!-- 'email' OR 'fax' OR 'print' -->
 </cyberPOSTRequest>
```

The settings selected by the user are converted to an SQL WHERE clause fragment and appended to the SQL statement that retrieves the list of recipients.

INDEX

architecture, 2
'Attach File', 8
browse, 6, 23, 24, 29
browsing, 24, 28
Cash Sale, 5, 28, 38
Cash Sale Return, 5
classpath, 9, 10
Compiled classes, 10
cpxml, 2, 3, 4, 5, 7, 10, 15, 18, 19, 22, 23, 26, 29, 31, 37
- **<admin>, 35**
- **<batchSendOrderBy>, 24**
- **<batchSendWhere>, 23**
- **<bindData>, 26**
- **<bindDataGroup>, 26, 27**
- **<body>, 19**
- **<browseOrderBy>, 24**
- **<browseTable>, 28**
- **<browseWhere>, 24**
- **<byPosition>, 29**
- **<byTags>, 29**
- **<charPos>, 29**
- **<column>, 28**
- **<comboItem>, 28**
- **<companySerNo>, 20**
- **<dataSource>, 20**
- **<default>, 26, 28**
- **<display>, 28**
- **<documentName>, 19**
- **<email>, 35**
- **<emailSettings>, 19**
- **<endDocumentTag>, 20**
- **<endTag>, 29**
- **<excludeInCompany>, 20**
- **<field>, 23, 24, 26, 28, 37, 38, 39**
- **<filename>, 20**
- **<findMasterWhere>, 23**
- **<findTransactionWhere>, 23**
- **<label>, 26**
- **<length>, 29**
- **<lineNo>, 29**
- **<logEntry>, 21**
- **<master>, 22**
- **<masterFrom>, 22**
- **<masterWhere>, 22**
- **<method>, 32**
- **<name>, 32**
- **<newline>, 26**
- **<onResult>, 32**
- **<orderByClauses>, 22**
- **<parameter>, 31**
- **<parseText>, 29**
- **<properties>, 19**
- **<report>, 20**
- **<reportParameters>, 31**
- **<result>, 32**
- **<robotWhere>, 23**
- **<sql>, 22, 24**
- **<startDocumentTag>, 20**
- **<startTag>, 29**
- **<subject>, 19**
- **<text>, 26**
- **<toolTipText>, 26**
- **<transaction>, 22**
- **<transactionFrom>, 23**
- **<transactionOrderBy>, 23**
- **<transactionWhere>, 23**
- **<value>, 28, 37**
- **<whereClauses>, 22, 23**

Credit Note, 5, 28, 37, 38
Crystal Reports, 2, 3
Custom method invocation, 32
custom methods, 10
data source, 2, 4, 11, 19, 20, 22, 29
database, 8, 9, 11, 12, 24, 26, 28, 32, 36, 37
dateBrowser, 29
debtor statements, 4, 19, 21, 22, 23
Document Combo Box, 5
Document Definition, 3, 5, 20
Document Parameters, 7
extra1, 22, 26, 27
fax, 8
- **monochrome logo, 10**

fields, 4, 7, 10, 11, 15, 20, 26, 29, 31
Fields, 15
fileBrowser, 29
filters, 7, 8, 23
HashMap, 10, 16, 32
HTML, 2, 10, 19
Invoice, 5, 28, 37, 38
jar, 9, 10
Jasper, 2, 3, 10, 16, 20, 21, 31, 37
Jasper Reports, 2, 3, 31, 37
Java, 11, 32, 36
JDBC, 9, 11, 12, 20, 22, 26, 29
JDBC Templates, 12
key1, 4, 19, 20, 21, 24, 26, 39
Linux, 5, 13, 17
log, 7, 8, 21, 36

logo, 10
multi-company, 12, 20, 37
One Of, 7
online update, 9
parameters, 3, 7, 10, 16, 31, 32, 36, 37
Pause, 8
preview, 9
Print, 9
Range, 7
return values, 32
robot, 2, 9, 10, 12, 17, 23, 24, 35, 36, 39
root element
  cyberPOST2, 18
sample application, 4, 5, 6, 7, 23
security, 24, 36
setup, 9, 10
  database, 11
  details, 10
  Fax, 13
  Mail Servers / Robot, 12
SMTP, 12
SQL, 3, 16, 22, 23, 24, 25, 29, 31, 39
Stop, 8
sub-documents, 5, 28
System variables, 15
templates, 2, 3, 10, 11, 12
text, 2, 4, 7, 8, 10, 15, 19, 20, 21, 22, 23, 26, 27, 28, 29, 30, 31
text parser, 20
unique keys, 6
User Defined Parameters, 14
Windows, 5, 13, 17
working directory, 5, 8, 9, 10, 12, 17, 19, 32
XPATH expressions, 15
Zigineeze, 2, 23, 36, 37, 38, 39
Zigineeze XML
  <companySerNo>, 20, 37
  <cyberPOSTRequest>, 37, 38, 39
  <document>, 37
  <field>, 37
  <replyBy>, 37