IDF - System-Link User Guide

Publication Information

Releases:

- LX Release level 06.03, build version 06.03.00.03.12
- System21 build version 05.03.00.03.09
- XA Release level 9, build version 02.09.00.03.21
- Server PTF level SH51084

Note: Much of the functionality and syntax described in this document will also be present in earlier build levels.

Publication date: October 22, 2014

Contents

About this guide

Consumer demand for the universal availability of information is growing; the expectation that information should be accessible from wherever it is needed, rather than only being available from a single, fixed source.

A business system must be able to support communication with as many disparate systems as possible: workstations, Internet browsers, mobile devices, or business-to-business transactions involving servers and Microsoft COM, Enterprise JavaBeans, CICS, SQL, HTTP, etc. IDF System-Link provides LX, System21, and XA the ability to participate in the world of intercomputing.

The global IDF Naming process (controlled via Link-Manager) includes and uses an embedded web server that will run on your System i as part of a normal installation and operation. As a result, the installation of a separate web server for use by System-Link or the Simulator is not required. The web server solution should meet the needs for the majority of installations; but, System-Link may still be configured to use an external web server such as IBM WebSphere Application Server for those organizations requiring a more extensive web server feature set.

Caution: Before using System-Link with Integrator objects or Infor ERP primary objects that have been extended with Integrator, you **must have** Infor XA Enterprise Integrator installed. These objects will not be usable through System-Link with Standard Integrator.

In the case of extended Infor ERP primary objects, **the entire object (not just its extension) will be unavailable** through System-Link without installing Enterprise Integrator.

Common Terms

For the purpose of this document:

- Infor ION 10.1 or later is referred to as ION.
- Infor LX is referred to as LX
- Infor System21 is referred to as System21
- Infor XA is referred to as XA.

Intended audience

This guide is intended for system administrators or technical personnel who want to implement integrations between IDF-enabled Infor ERP systems and external business applications.

This guide assumes a working knowledge of XML, which is beyond the scope of this guide.

Related documents

You can find the documents in the product documentation section of the Infor Xtreme Support portal, as described in "Contacting Infor".

- *System-Link MQ Installation and Configuration Guide* - If you intend to use System-Link MQ (System-Link via WebSphere MQ)
- *IDF System-Link Reference Guide* - a description of the IDF System-Link request and response element hierarchy.

Contacting Infor

If you have questions about Infor products, go to the Infor Xtreme Support portal at www.infor.com/inforxtreme.

If we update this document after the product release, we will post the new version on this Web site. We recommend that you check this Web site periodically for updated documentation.

If you have comments about Infor documentation, contact documentation@infor.com

Chapter 1 What is System-Link?

1

System-Link is a bridge between outside systems and IDF-enabled ERP systems (LX, System21, and XA). The term *outside system* could mean one or more of any number of programs and platforms, including:

- A web page
- An RPG program
- A Java program on a System i or an auxiliary machine
- Other systems and processes using ION
- Other ERP systems on other platforms

The calling program can be any system that can send and receive XML. As long as a system can get its XML to System-Link it can interoperate with IDF. It should come as no surprise, therefore, that there are a number of ways to call in to System-Link. For example:

- A web page (HTML) or Java program would communicate via your web server.
- An RPG program can communicate via your web server or directly using a local program call.
- ION-enabled products would communicate via ION using business object documents (BODs).
- Other ERP systems may interact with each other and System-Link via WebSphere MQ.

In short, the way System-Link is called may depend on language, platform, and business needs. RPG, HTML, and Java may each have a preferred technique for calling System-Link. When different platforms are involved, the communications link (LAN, internet, WebSphere MQ, ION, etc.) may dictate how System-Link is called. It is not important that you call System-Link in the right way, because there often is no one right way. It is important to understand the calling options available and the relative advantages of each.

At a high level, there are four ways to call System-Link: the local program interface (LPI), the web server, a message queuing system, and ION.

System-Link Components

Before taking a look at the ways to call System-Link, you should be familiar with the components that make up System-Link. System-Link includes these five components):

- SLS (System-Link Server): This component contains the core functions required for System-Link to operate. By default, when System-Link is licensed and installed, the host system includes one System-Link Server process. If you need to distribute the System-Link processing more widely on the host system, you can create additional instances of the System-Link Server process. System-Link Server processes can be located on auxiliary machines.

- SLC (System-Link Controller): This component receives requests for System-Link and directs the requests to the appropriate servers. You can have only one controller per environment, but you can have more than one server.
- SLI (System-Link Inbox): This component routes requests from the global web server to SLC or the IDF inbox. SLI can appear in the list of processes for an associated XA environment; but the global environment controls starting and stopping the System-Link Inbox process instance. If a System-Link request comes in, depending on the type, and System-Link is not active, then the request will be put in the IDF inbox.
- SLA (System-Link Adapter): This component allows inbound and outbound communication through System-Link to other Infor products using the IDF Inbox. This process must be running for an associated XA environment to be accessible by Infor IDF Inbox. The System-Link Controller and one or more System-Link Server instances must be running before the System-Link Adapter can deliver inbound messages for processing.
- SLO (System-Link Outbox): This component contains the functions that provide resiliency for the response forwarding feature in System-Link. If System-Link cannot forward a copy of the response to one or more destinations, System-Link adds the copy to the outbox. SLO regularly checks the outbox and attempts to resend. SLO is discussed under “System-Link Outbox process”.

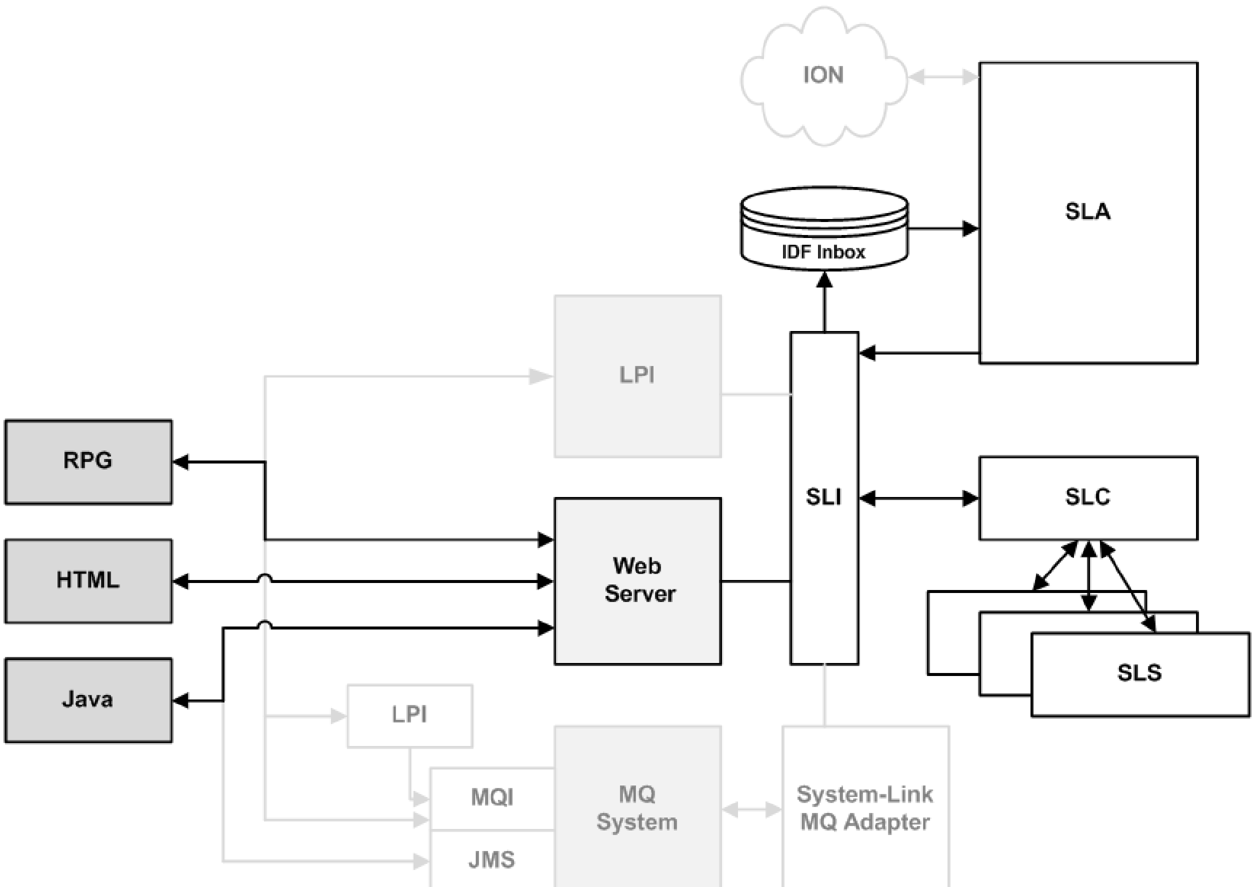
Communicating with System-Link

As stated earlier, there are four ways to call System-Link: the web server, the local program interface (LPI), ION, and WebSphere MQ.

Web Server

One of the easiest ways to use System-Link - and one of the quickest ways to obtain business benefits - is to use it with web pages. It is very easy to create a secure web page to inquire into LX, System21, or XA data. Many companies supply customers, vendors, and partners with a user ID and password so they can get up-to-the-minute transaction data from the web site of the company. In such a scenario, the communication between the web page and System-Link would be through the web server. A company could access System-Link from the web server using programs written in RPG or Java.

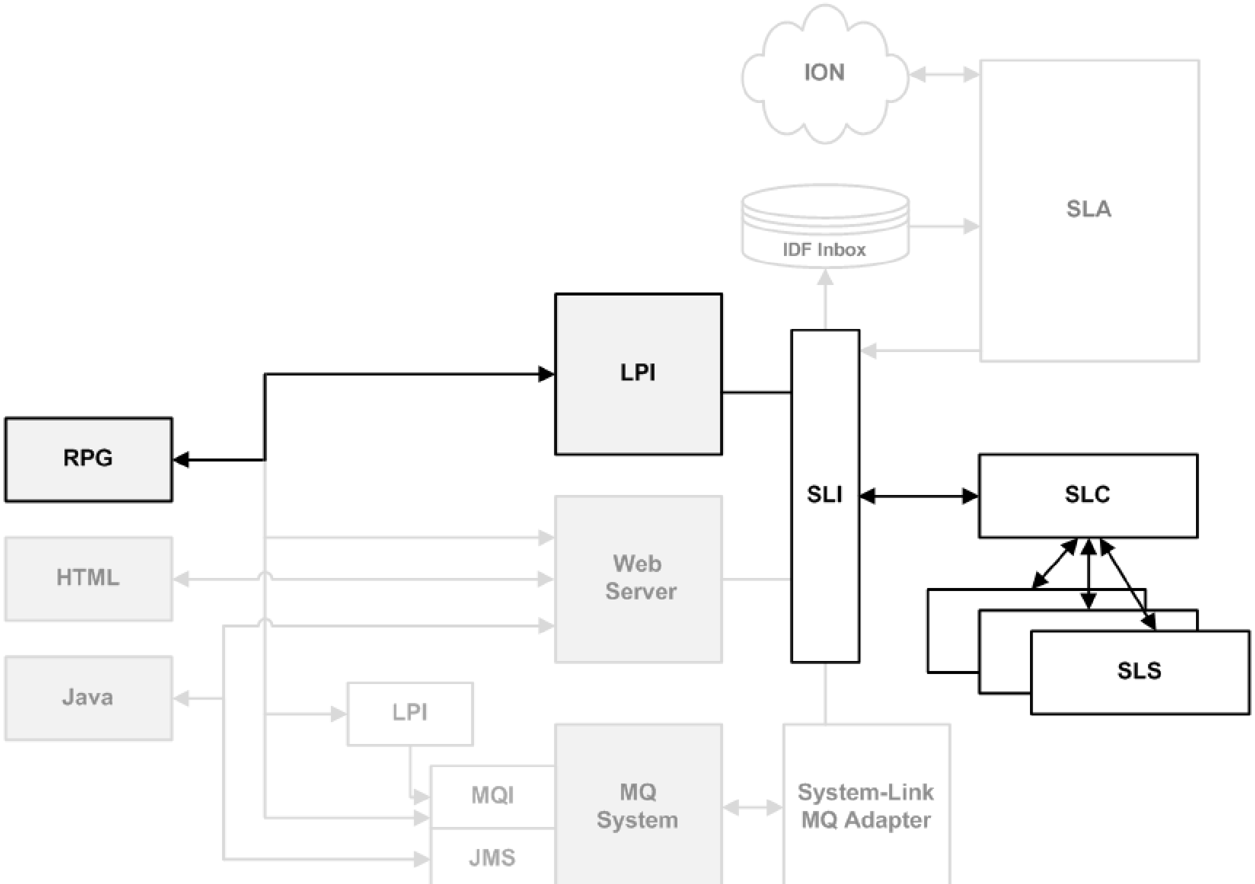
The diagram below illustrates how different technologies might interact with System-Link via Web Server.



Local Program Interface

Often companies wish to interact with IDF using their own RPG programs. In such a case, using the web server is not required. The LPI allows an RPG program to communicate directly with System-Link.

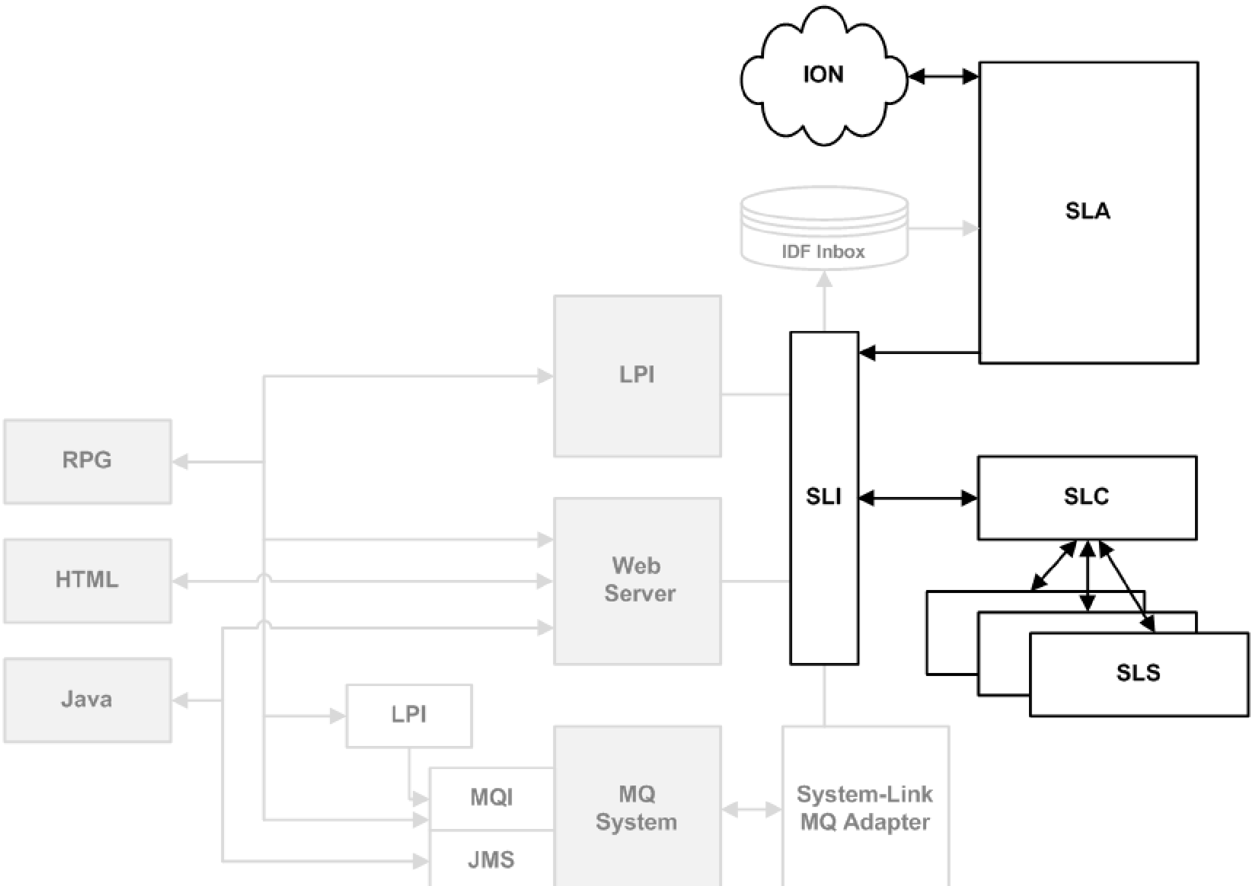
The diagram below illustrates how RPG programs interact with System-Link using LPI.



ION

Infor Intelligent Option Network (ION) is the Infor messaging middleware solution, accessible by both Infor and third-party software applications. Although we will cover some aspects related to IDF-ION integration, the complete details of interfacing applications to ION are beyond the scope of this document. Consult the related ION integration documents on the Infor Xtreme Support portal.

The diagram below illustrates how products connected via ION will interact with System-Link:



WebSphere MQ

Note: Although this section was created specifically to support WebSphere MQ, it is possible to use other message queuing systems that support JMS.

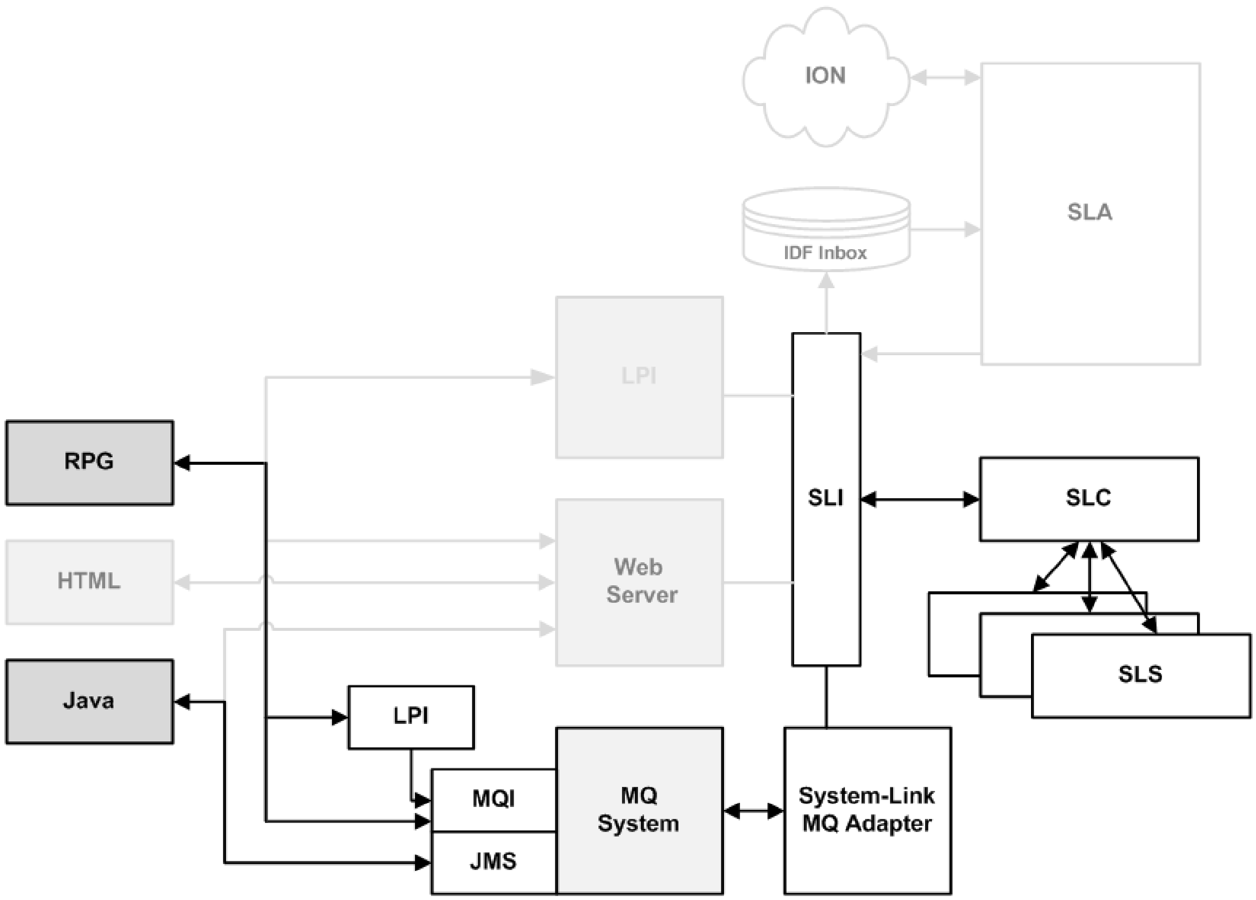
A message queuing system provides a communications layer allowing synchronous or asynchronous communications between multiple processes. A program using WebSphere MQ can interact with System-Link by placing a request on a specific request queue and then awaiting the response on its own response queue.

The method with which a program interacts with System-Link MQ depends on the language of the program:

- Java programs can access WebSphere MQ directly via JMS
- RPG programs can access WebSphere MQ directly via MQI (provided by IBM)
- RPG programs may use System-Link MQ without dealing directly with WebSphere MQ by using the Local Program Interface (LPI) provided by Infor. From the perspective of the programmer, this interface is identical to the LPI described earlier. The difference is that one server program calls System-Link directly, while the other server program calls System-Link via WebSphere MQ.

All of these methods yield similar results, so the decision is one of programmer preference. Programmers who already have WebSphere MQ and are familiar with MQI or JMS may prefer those interfaces. Programmers who already use the LPI to communicate to System-Link directly may prefer to use the same interface to communicate with WebSphere MQ.

This diagram illustrates how RPG or Java programs might interact with System-Link using Websphere MQ:



Related IDF Programs

Link Manager is the GUI-driven facility that allows a company to distribute the server load across a variety of system resources. These system resources include the System i, IXS cards, and LPARs in the System i, Windows, Linux servers on the LAN, etc. Link Manager also controls all IDF Java environment servers, as well as the deployment of the System-Link MQ Adapter, which serves as a bridge between WebSphere MQ and System-Link.

Power-Link is the GUI-based interface to IDF for designers. Part of the Power-Link functionality is a System-Link request builder tool, found under the **Navigation** menu on most action-oriented windows (for example, list browser, object browser, create dialog, etc.). The request builder saves time compared with creating a request entirely from scratch and is recommended for use whenever possible.

Chapter 2 Accessing System-Link via a Web Server

2

The global IDF Naming (NAM) process, controlled by Link Manager, has an embedded web server that is installed during the normal installation process. This web server runs on your System i, and can be accessed by the base URL *http://<your System i name>:<your secondary port>*, where

- *<your System i name>* is the name of your System i such as “my400.mycompany.com”
- *<your secondary port>* is the secondary port number for the client framework. Unless you have explicitly changed the port during installation, this value will be “36001”. (If required, the secondary port number can be validated by consulting the QPRINT files associated with the MPX processes started by the UJOB. The port number will appear as a parameter to the processes.)

Therefore, the base URL for the above example would be *http://my400.mycompany.com:36001*.

Interfacing with the System-Link servlet via HTTP is done by passing your System-Link request in the body of a POST message to *http://<your System i name>:<your secondary port>/SystemLink/servlet/SystemLinkServlet*. The message body contains a single key-value pair: the key “SystemLinkRequest” and the value consisting of your System-Link request document. (Alternately, if the key “SystemLinkRequestURL” is used, the value should consist of the URL where your System-Link request document can be obtained.)

An example web page using the System-Link servlet is the System-Link Simulator (a request testing and validation tool that is described later in this guide), which can be found at *http://<your System i name>:<your secondary port>/SystemLink*. The source for this web page can be found in the IFS folder */XA/Web/SystemLink*.

The embedded web server is considered the default for System-Link, and is designed to provide a turnkey solution that will meet the needs of the majority of installations. If your requirements dictate that a separate web server be used such as IBM WebSphere Application Server, then the System-Link servlet can be run in that alternate web server. The installation and configuration of an alternate web server is beyond the scope of this document. Consult the information provided by your web server vendor, as well as the installation guides on Xtreme.

Chapter 3 Accessing System-Link via the Local Program Interface

3

Overview

System-Link provides an XML-based interface to the LX, System21, and XA applications. This enables applications to interact with these applications via XML from a variety of platforms. The System-Link Local Program Interface (LPI) makes it possible to perform this same integration from server applications without requiring communications through a web server or WebSphere MQ.

The same System-Link request that the server programmer would send to the web server can be sent instead to the LPI. The LPI will return the same System-Link response that the web server would return. The only difference is that the System-Link documents are passed via program parameters.

LPI Comparison

There are two RPG programs that provide implementations of the Local Program Interface: one that interfaces with System-Link directly and one that interfaces with System-Link via WebSphere MQ. These comments describe their differences:

1. **LPI (direct)** – This program makes a direct call to System-Link for environments on the same System i host. No installation of WebSphere MQ is required. The communication is entirely synchronous: the System-Link Server associated with the requested environment must be running at the time that the call is made. If the server is not available, an error will be returned.

2. **LPI (via MQ)** – This program communicates with System-Link via WebSphere MQ and the System-Link MQ Adapter. This version requires the installation and configuration of WebSphere MQ and Directory Services. The communication can be asynchronous in that the System-Link Server associated with the requested environment does not have to be running at the time the call is made. But, if the server itself is not available, the caller will block until the request completes.

The LPI (direct) program has the capability of sending System-Link requests to environments hosted on a different System i. This feature is described later in this chapter.

Note: If you intend to use LPI via MQ, see “Accessing System-Link via System-Link MQ”. It contains further relevant installation and usage information.

Interacting with the Local Program Interface

LPI usage involves:

- The application program creating a System-Link request.
- The LPI being called with the System-Link request as a parameter. The size of the request parameter is limited to 64K characters.
- The System-Link Server processing the request and returning the System-Link response to the LPI.
- The LPI sending the System-Link back to the application program. A message is returned if an error condition is encountered.

Program parameters

The program name of the Local Program Interface is PSVPSR1R. Its parameters include:

Parameter	Description	Usage	Size	Type
P#XREQ	System-Link request	I	65,535	A
P#XRSP	System-Link response	O	65,535	A
P#MRQR	More query results (*YES/*NO)	O	8	A
P#ERFD	Internal errors (*YES/*NO)	O	8	A
P#MSID	External errors (PSX0122/PSX0130)	O	7	AV

P#XREQ This is the System-Link request. The request cannot be longer than 64K bytes.

P#XRSP This is the System-Link response. If the response is longer than 64K bytes, *PSX0130* is returned indicating that the response was too large.

P#MRQR A return value of **YES* indicates that there are more records than what have been returned so far. **Note:** For this parameter to ever return **YES* your request must include *retainResult='true'*.

P#ERFD When the System-Link response contains the <*Message type*=*error*> element, this parameter is set to **YES*. This indicates that the response contains error messages.

P#MSID There are three values for this parameter:

- **(blank)** The request was processed normally.
- **PSX0122** There was an error in processing. For example, the System-Link server was not started.

PSX0130 The response was too large. You should change your request in order to reduce the response. For example, insert a *maxReturned* value when requesting a list of records.

Example 1: Request

The following XML represents a request to change the quantity of requisition 1234 to 101.000 in environment P0. (The user ID is shown as UUUUUUUUU and the password is shown as PPPPPPPPP.)

```
<?xml version='1.0' encoding='ISO-8859-1'?>

<!DOCTYPE System-Link SYSTEM 'SystemLinkRequest.dtd'>

<System-Link>

   <Login userId='UUUUUUUUUU' password='PPPPPPPPPP'
          maxIdle='900000'
          properties='com.pjx.cas.domain.EnvironmentId=P0,
                      com.pjx.cas.domain.SystemName=USATLD06,
                      com.pjx.cas.user.LanguageId=en'/>


   <Request sessionHandle='*current workHandle='*new'
            broker='EJB' maxIdle='900000'>

     <Update name='update_req_1'
             domainClass='com.mapics.pm.Requisition'>

       <DomainEntity>

         <Key>

            <Property path='requisition'>

               <Value><![CDATA[1234]]></Value>

           </Property>

         </Key>

         <Property path='quantity'>

           <Value><![CDATA[101.000]]></Value>

        </Property>

       </DomainEntity>

     </Update>

   </Request>

</System-Link>
```

To send this request to the Local Program Interface, this command would be used:

```
call psvpsr1r ('<?xml version="1.0" encoding="ISO-8859-1"?><!DOCTYPE System-
Link SYSTEM "SystemLinkRequest.dtd"> <System-Link> <Login userId="UUUUUUUUUU"
password="PPPPPPPPPP" maxIdle="900000"
properties="com.pjx.cas.domain.EnvironmentId=P0,
com.pjx.cas.domain.SystemName=USATLD06, com.pjx.cas .user.LanguageId=en"/>
<Request sessionHandle="*current" workHandle="*new" broker="EJB"
maxIdle="900000">< Update name="update_req_1"
domainClass="com.mapics.pm.Requisition"> <DomainEntity> <Key> <Property path="
requisition"> <Value><![CDATA[1234]]></Value></Property></Key><Property
path="quantity"><Value><![CDATA[101 .000]]></Value> </Property>
</DomainEntity> </Update> </Request> </System-Link>' ' ' ' ' ' ' ' ')
```

This command can be described in sections.

Call the program

```
call psvpsr1r
```

This is the call to the program.

First parameter: *P#XREQ*

```
call psvpsr1r ('<?xml version= "1.0"...</System-Link>'
```

The first parameter is the System-Link request as shown above. Most of it has been removed, as shown by the ellipsis (…) to simplify the illustration.

Single quotes define the parameter. **Note:** Since the System-Link request also contains quotes, they are represented as double-quotes. For example, there are double-quotes around the version (“1.0”) versus the single quotes (‘1.0’) in the System-Link request at the beginning of the examples topic. Double-quotes need to be used in the XML when it is inside a parameter. The LPI will change them to single quotes when it passes the request to System-Link.

The remaining parameters are all blank: four sets of quote-blank-quotes separated by blanks.

```
call psvpsr1r ('<?xml version= "1.0"...</System-Link>' ' ' ' ' ' ' ' ')
```

Example 2: Response

If the requisition is found and there are no errors, this System-Link response is returned. (Note that some attributes on the System-Link root element and Response element have been omitted for clarity.)

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<System-Link version="1.0" hostVersion="1.0">

  <LoginResponse actionSucceeded="true">

    <SessionHandle value="-6816b5eb:f0aa27ee6b:-7e6f"/>

  </LoginResponse>

  <Response sessionHandle="-6816b5eb:f0aa27ee6b:-7e88"
            workHandle="-6816b5eb:f0aa27ee6b:-7e6d"hasErrors="false"
            hasWarnings="false">

     <UpdateResponse name="update_req_1"
                     requestedDomainClass="com.mapics.pm.Requisition"
                     actionSucceeded="true"/>

  </Response>

</System-Link>
```

The returned parameters would be:

Parameter	Value	Description
P#XRSP	<?xml version=“1.0” … </System-Link>	This is the returned System-Link response
P#MRQR	*NO	The response is complete
P#ERFD	*NO	There are no internal errors
P#MSID	(blank)	The request was processed normally

Example 3: Response with errors

Sometimes a request will generate an error condition. For example, if the requisition number in our example did not exist, the following is System-Link response would be returned. (Note that some attributes on the System-Link root element and Response element have been omitted for clarity.)

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<System-Link version="1.0" hostVersion="1.0">

   <LoginResponse>

      <SessionHandle value="-6816b5eb:f0aa27ee6b:-7e4c"/>

   </LoginResponse>

   <Response sessionHandle="-6816b5eb:f0aa27ee6b:-7e88"
            workHandle="-6816b5eb:f0aa27ee6b:-7e4a" hasErrors="true"
            hasWarnings="false">

      <UpdateResponse name="updateObject_Requisition"
                      requestedDomainClass="com.mapics.pm.Requsition"
                      actionSucceeded="false">

         <Exception name="com.pjx.eScript.RequestDataException">

            <Text><![CDATA[ Error occurred during update]]></Text>

            <Message type="error">

               <Text><![CDATA[ Error occurred during update of
      com.mapics.pm.Requisition object]]></Text>

            </Message>

            <Message type="error">

               <Text><![CDATA[ (E) Requisition 1234 no longer
      exists.]]></Text>

               <DetailedText><![CDATA[Explanation . . :   The record you
      are trying to change or delete no longer exists in the database. It
      might have been deleted by another user or job. Your action: Refresh
      your screen and the record should disappear from the list. If not
      please contact your XA support representative.]]></DetailedText>

            </Message>

            <Message type="error">

               <Text><![CDATA[ (E) The requisition record could not be
      locked.]]></Text>

               <DetailedText><![CDATA[Explanation . . : You attempted to
      access a requisition record, but the record is being maintained by
```

```
      someone else. Your action . . : Select another requisition to work
      with or try again.]]></DetailedText>

            </Message>

            <Message type="information">

               <Text><![CDATA[ Check for XML errors]]></Text>

               <DetailedText><![CDATA[ Check your XML for
      errors]]></DetailedText>

            </Message>

         </Exception>

         <Exception name="com.pjx.eScript.TransactionException">

            <Text><![CDATA[ Transaction failed due to nested
      exceptions]]></Text>

            <Message type="information">

               <Text><![CDATA[ The transaction failed due to the nested
      exceptions indicated.]]></Text>

            </Message>

         </Exception>

      </UpdateResponse>

   </Response>

</System-Link>
```

The returned parameters would be:

Parameter	Value	Description
P#XRSP	<?xml version=“1.0” … </System-Link>	This is the returned System-Link response
P#MRQR	*YES	The response is complete
P#ERFD	*YES	There are internal errors
P#MSID	(blank)	The request was processed normally

You would need to parse the System-Link response in order to find the errors. Parsing XML is described in more detail below.

Programming to the Local Program Interface (LPI)

One of the input parameters to the LPI is the System-Link request. The LPI passes the System-Link request to System-Link where it is processed. System-Link returns a System-Link response to the LPI, which passes it back to the calling program in an output parameter (Figure A).

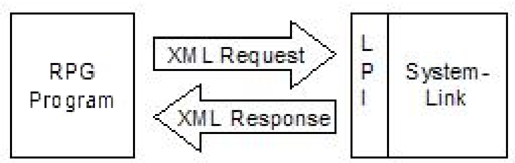


Figure A

Creating these programs involves two steps that may be new to an RPG programmer. First, the System-Link request must be assembled. Power-Link provides tools to help create the System-Link request. An option on the **Navigation** menu provides the System-Link request for the list or card file being displayed, and can be changed and tested, if necessary, using the System-Link Simulator. The request must contain four items that may or may not be variables: the user ID, password, environment, and language. In addition, there will usually be one or more input variables, such as customer, vendor, or item. All variables must be included in the System-Link request that is sent to the LPI.

Second, the results must be parsed. The RPG program must be able to extract data from the XML string and place it in variables in the RPG program.

Multiple techniques can be used to assemble the System-Link request and parse the System-Link response. RPGLE contains features that facilitate working with string data. We have written some programs to assemble the System-Link request using messages. We have written another program to illustrate how the response can be parsed. These programs can be downloaded[1]. The remainder of this chapter describes these programs.

[1] These are not supported programs and are provided for illustration purposes only.

Example programs

To illustrate how the LPI can be used we will use a set of programs that print a list of POs for a vendor. The vendor number is an input parameter of the RPG program, which will request a maximum of ten records. If the list is not complete, LPI will request up to five more, and continue doing so until all the records have been returned. LPI will then print the results.

You can download these programs (refer to Figure B):

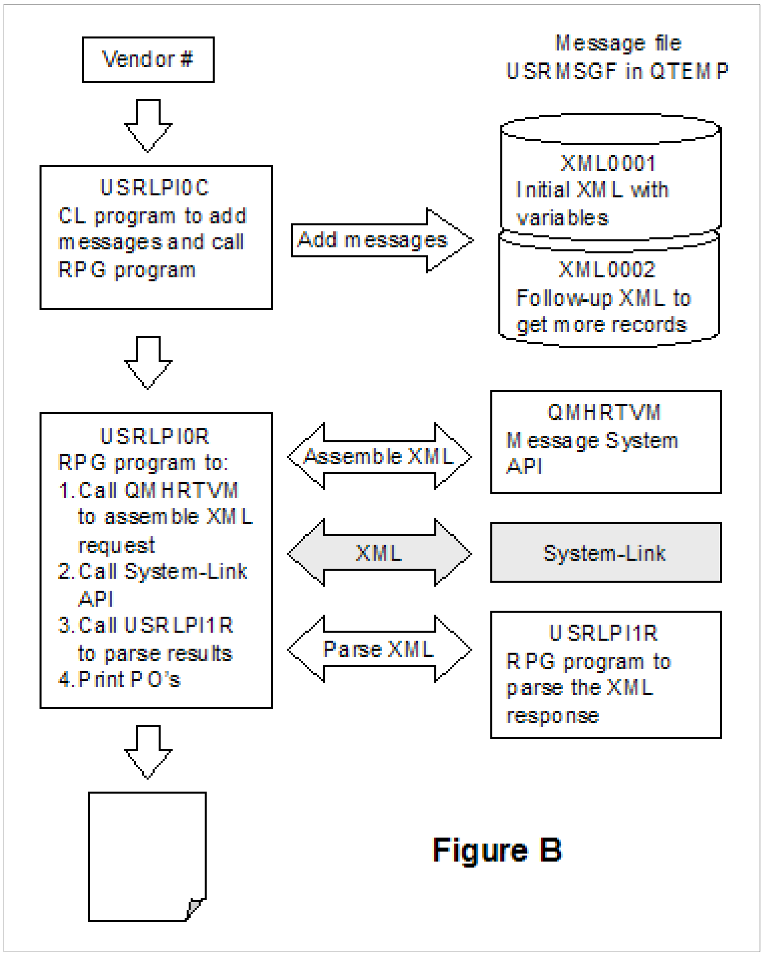


Figure B

URSLPI0C

NOTE: You may choose to download these programs, make some necessary changes, and then run them. To simplify that process we have provided this CL program. The CL program will create the messages you need during runtime. Should you choose to employ this message file technique, you would create permanent messages, and a CL program like this one would not be necessary. This chapter describes what the supplied programs do, and then describes what you would do differently to use the same techniques in production.

This CL program writes two messages to a message file in QTEMP. The first is the QueryList request https://www.inforxtreme.com/esknowbase/root/DLPublic/40531/SH15992.pdf is the initial request for POs. This request contains five variables, one of which is the input parameter (for example, the vendor number). The other four are user ID, password, environment, and system name. We will discuss how you can enter your own values for these later.

The second message is a FetchNext request that will be used to get additional POs (if any). This request contains three variables: the session, work handles, and the maximum number of records to be retrieved.

USRLPI0R

This is an RPG program that calls the LPI. The PRG program can call the direct LPI or the MQ Series LPI, which is a different program with the same parameters[2]. This program also receives the System-Link response and prints the POs.

Assembling the System-Link Request

One of the parameters of the LPI is *P#XREQ* which holds the System-Link request. It must be a complete request, without any substitution variables. Thus, one of the functions of this program is to replace variables in the System-Link request with actual values. Since the XML is stored in a message, we can use *QMHRTVM*. This is a system API that can retrieve the message from the message file and replace substitution variables with data. This is exactly what we want to do to assemble our System-Link request.

The parameters for *QMHRTVM* include the message ID (which contains the System-Link request in the first message created by our CL program) and the message data. Our program sets the message data to these predefined values:

User ID	USER
Password	PASSWORD
Environment	MM
System name	SYSTEM

The data for the fifth substitution variable (the vendor) is supplied by an input parameter. *QMHRTVM* returns the XML string that is put into *P$XREQ*. This is the parameter that will be passed to the LPI.

Calling the LPI

As stated earlier, there are two different LPIs with the same parameters: one for calling System-Link directly and one for calling System-Link via WebSphere MQ. This program has a variable called LPI Option (*wLPIOpt*) that can be set to one of three values:

SystemLink	Call *PSVPSR1R* (the direct LPI)
MQSeries	Call *PSVSMI1R* (the WebSphere MQ LPI)
MQIsample	For use with the LPI provided by IBM (see program *USRLPI2R*, discussed below)

Based on the value of the LPI Option, the appropriate LPI is called.

[2] There is also an MQ interface (MQI) provided by IBM. See discussion of program USRLPI2R.

Getting more records

The System-Link request is set to retrieve a maximum of ten records. You can increase this by simply replacing the *maxReturned* value in the System-Link request with whatever value you choose. If you need to retrieve additional records, then you will need to use the FetchNext request.

In order to perform the FetchNext request, the parsing program (*USRLPI1R* – discussed below) must be called. The parsing program will retrieve the session and work handles, and pass them back to *USRLPI0R*, which then uses the same technique described above to assemble the System-Link request. *QMHRTVM* is called again, but this time the message ID is the second message created by our CL program, and the data to be inserted is the session and work handles, as well as a new *maxReturned* value (five records instead of the original ten).

Printing the data

The program prints the data in the normal way, but to be able to print the data, it must be able to recognize it. The System-Link response is a long string of character data. In order to print a PO number, for example, our program needs to be able to extract it from the XML string. To do this, it calls *USRLPI1R*.

USRLPI1R

This program consists of a series of subroutines. The *USRLPI0R* program calls USRLPI1R when it needs some data. For example, it will call USRLPI1R to get:

- the first (or next) domain entity, which is a record
- the first (or next) key field for that domain entity
- the first (or next) non-key field for that domain entity, etc.

The program is called field by field, one at a time. In addition to business data, this program is also called to retrieve the session and work handles needed for the FetchNext request.

Because the System-Link response is a long string of character data, the parsing program makes extensive use of scan and substring functions. Every scan is based on character positions in the System-Link response. The beginning and end elements are stored as constants for use in the scan operation.

This simplified example will illustrate how the parsing function works.

Function: get domain class for the first domain entity (from subroutine *GetDmnEnt*)

Step 1: Starting at position 1, scan the System-Link response (*p#xrsp*) looking for the domain entity element (*DmnBegS*). The resulting variable (*wPos*) will hold the starting position of the domain entity element. (The starting position for the scan is stored in a field (*cDmnNext*) and increased as additional records are read.)

Step 2: Add 27 to the position (wPos + 27). This is because the constant representing the domain entity element is 27 positions long. Place the result in a variable (*wStr*) that represents the first character of the data you want. In this case, that data is the domain class “com.mapics.pm.PurchaseOrder”.

Step 3: Starting from the result in step 2 (that is, the first position of the data you want) scan the System-Link response (*p#xrsp*) for the domain entity end element (*DmnBegE*). The resulting variable (*wEnd*) will hold the starting position of the end element.

Step 4: Now you have the starting position of the data you want and the starting position of the end element (which is the first position after the data). Subtracting one from the other gives you the length of the data, so you can use the substring function to place the data in the domain class variable (*p#dmcl*).

All the parsing uses this same technique. The result is that the substring of the XML is placed in a program variable where it can be manipulated by RPG.

One item to note is that the XML data is all character data. This means that a numeric value in the database, such as a quantity, appears as an alphanumeric value in the System-Link response. It needs to be changed back before it is stored in the variable. To do this, the *atof* function is used.

Using the sample programs as a model

The sample programs are provided to you for use as a model to create your own programs. You will need to:

- **Create your own messages.** There is an option on the **Navigation** menu to give you the System-Link request for whatever view or card file you have displayed. You can use the Simulator to adjust that System-Link request, or create your own, and test it. Once you have the System-Link request you want, you can save it in a permanent message. Consider whether you want to perform requests in ‘bunches’ (that is using the FetchNext technique), in which case you need a pair of messages, one for the initial query and one message for additional records. For example, if you have some logic to determine whether additional records are needed, you would want to perform requests in bunches. . Alternatively, you can request all records with the initial request (*maxReturned =’ -1’*).

Caution: Retrieving a large amount of information in one request may exceed the size of the LPI buffer…64K. Use this feature with caution.

- **Build your request.** Use the *USRLPI0R* program as a model. You may want to make the sign-on parameters as being of type ‘information’. Be sure the switch is set the way you want it (that is, to call System-Link directly). Check your message data parameters to ensure that they match what is present in your request.
- **Parse the results.** There are two basic approaches to parsing XML data. One makes a pass through the XML and processes the desired fields one-by-one as they appear. Our sample programs use this single-step method. A second approach has two steps: it first makes a pass through the XML and simply places all fields encountered into a table. The desired field information is then extracted from the resulting table for processing in the second step. One advantage of this two-step technique is that you can write the XML parsing program (step one) once and reuse it in different programs requiring different field values. The disadvantage is that it can be inefficient, such as when you have a large number of fields (resulting in a large table) and only want the first one.
- **Process the response.** Program *USRLPI0R* shows how the data can be parsed (by calling the parsing program) and printed as required. You may want to write the data to a file, or do something else with it.

Special notes on sample program USRLPI2R

This sample program is a simplified implementation of LPI using System-Link MQ. System-Link MQ uses WebSphere MQ as a communication medium between the caller (in this case the LPI program) and System-Link. *USRLPI2R* is based on a sample program provided by IBM that uses their Message Queue Interface (MQI):

Program name	AMQ3REQ4
Source file name	QRPGLESRC
Library name	QMQMSAMP

Using this program as a starting point, we have made some changes to conform to the specific requirements of System-Link MQ. The resulting program is *USRLPI2R*. These changes include:

- Queue name - The System-Link queue name is *XSA.INPUT.QUEUE*. This cannot be changed.
- Queue manager name - The System-Link queue manager name is *COM.PSI.MAINQM*. This cannot be changed.
- Work variables - We have provided as work variables the Unicode expressions for some of the System-Link elements.
- Parameters - We have defined the LPI parameters *P#XREQ*, *P#XRSP*, etc. and converted them to Unicode.
- Print to log - We have changed the printing technique to conform to the *USRLPI0R* program.
- Mainline - We have divided the four main sections into subroutines and then added the mainline program to call the subroutines. We did this to make the program easier to comprehend for the programmer.
- Evaluating the response - We added a section to evaluate the System-Link response to determine whether there are more records to query or any errors.

Other than these changes, the program should look very similar to the IBM sample program.

For further information on using System-Link MQ, consult “Accessing System-Link via System-Link MQ”.

Sending remote requests

Some integration deployments may require that actions originating in one environment send System-Link requests to another environment on a different System i. Two long-standing means of supporting these remote requests have been to:

- Have the calling program send the requests directly via HTTP to the remote host’s System-Link servlet.
- Use the System-Link response forwarding mechanism to send the results of a locally-run request to one or more remote destinations.

To simplify the routing task and to hide the complexities of HTTP in RPG, LPI now provides an updated program supporting requests for remote environments running System-Link. With the exception of one additional parameter used to specify the URL of the remote System-Link servlet, the new program interface has the same external behavior as the existing LPI (direct) program, and supports all of the existing LPI (direct) functionality as well.

Program parameters

The name of the updated program is *PSVRPI1R*. Its parameters and their semantics are identical to that of *PSVPSR1R*, with the inclusion of an additional parameter *P#URL*.

Parameter	Description	Usage	Size	Type
P#URL	URL for the remote System-Link servlet	I	256	A
P#XREQ	System-Link request	I	65,535	A
P#XRSP	System-Link response	O	65,535	A
P#MRQR	More query results (*YES/*NO)	O	8	A
P#ERFD	Internal errors (*YES/*NO)	O	8	A
P#MSID	External errors (PSX0122/PSX0130)	O	7	AV

P#URL This is the URL for the remote System i host's System-Link servlet. If a blank value is specified, then the behavior will be the same as with LPI (direct): the program will interpret a blank URL value to indicate that the request is for an environment on the local host.

P#XREQ This is the System-Link request. The request cannot be longer than 64K bytes.

P#XRSP This is the System-Link response. If the response is longer than 64K bytes *PSX0130* is returned indicating that the response was too large.

P#MRQR A return value of **YES* indicates that there are more records than what have been returned so far. Note: for this parameter to ever return **YES* your request must include *retainResult='true'*.

P#ERFD When the System-Link response contains the *<Message type=error>* element, this parameter is set to **YES*. This indicates that the response contains error messages.

P#MSID There are three values for this parameter:

- **(blank)** The request was processed normally.
- **PSX0122** There was an error in processing. For example, the System-Link Server was not started.
- **PSX0130** The response was too large. You should change your request in order to reduce the response. For example, insert a *maxReturned* value when requesting a list of records.

Example: calling PSVRPI1R:

The only difference between calling *PSVRPI1R* and calling *PSVPSR1R* is the URL provided as the first parameter.

If one wished to send a request similar to that mentioned in “Example 1: Request”, but instead to the System-Link server for environment “YY” on System i host “remotehost.mycorp.com” (and assuming that the standard secondary port is used), the corresponding command would be:

```
call psvrpi1r
('http://remotehost.mycorp.com:36001/SystemLink/servlet/SystemLinkServlet'
'<?xml version="1.0" encoding="'ISO-8859-1"?><!DOCTYPE System-Link SYSTEM
"SystemLinkRequest.dtd"> <System-Link> <Login userId="UUUUUUUUUU"
password="PPPPPPPPPP" maxIdle="900000"
properties="com.pjx.cas.domain.EnvironmentId=YY,
com.pjx.cas.domain.SystemName=remotehost.mycorp.com, com.pjx.cas
.user.LanguageId=en"/> <Request sessionHandle="*current" workHandle="*new"
broker="EJB" maxIdle="900000">< Update name="update_req_1"
domainClass="com.mapics.pm.Requisition"> <DomainEntity> <Key> <Property path="
requisition"> <Value><![CDATA[1234]]></Value></Property></Key><Property
path="quantity"><Value><![CDATA[101 .000]]></Value> </Property>
</DomainEntity> </Update> </Request> </System-Link>' ' ' ' ' ' ' ' ')
```

Note that aside from the different environment and host name in the System-Link request itself, the only change from before is the introduction of the URL for the remote System-Link servlet:

- http://remotehost.mycorp.com:36001/SystemLink/servlet/SystemLinkServlet

Past this point, interfacing with the updated program is the same as interfacing with the legacy one for LPI (direct).

Chapter 4 Accessing System-Link via System-Link MQ

4

System-Link MQ allows an outside program to send System-Link requests and receive System-Link responses via queues in WebSphere MQ (formerly MQ Series). There are three primary steps to using System-Link MQ:

1. Installation and configuration of WebSphere MQ and IBM Directory Services
2. Testing your System-Link MQ configuration
3. Designing and implementing the code in your program that interfaces with WebSphere MQ

The first two steps are covered in detail in the *System-Link MQ Installation and Configuration Guide,* follow that guide prior to continuing with step 3.

How you proceed with the third step – design and implementation – depends both upon how you intend to interface with MQ and upon the language in which your program will be written:

- If your program is written in Java, then you will be calling WebSphere MQ directly, using its JMS interface.
- If your program is written in RPG, then you can use the LPI (via MQ) program provided or can call WebSphere MQ directly using its MQI interface.

If you intend to use LPI via MQ as your program interface to System-Link MQ, then the actual interface to WebSphere MQ (as well as the System-Link MQ details) will be hidden from your program. See “Accessing System-Link via the Local Program Interface” for implementation details.

General Interface Program Flow

Whether you are using JMQ or MQI to interface with System-Link MQ, the basic overall steps will be the same:

1. Create a connection and a session with the queue manager/queue connection factory.
2. Obtain a reference to the System-Link request queue.
3. Create a new message structure for your request.
4. Create a temporary/dynamic queue to receive your response and place a reference to this in the message header.
5. Insert your System-Link request into the message structure.
6. Place the message on the System-Link request queue.
7. Wait for a response message on the response queue that you created in step 4.

8 Read your System-Link response from the response message.

9 Delete your temporary/dynamic response queue (explicit for JMS, implicit for MQI).

10 End your session and connection.

Calling System-Link MQ via JMS

JMS is a Java language standard interface supported by the major message queuing systems (including WebSphere MQ). General information, tutorials, and examples on using JMS can be found at http://www.oracle.com/technetwork/java/index.html, and are beyond the scope of this guide. One of the downloadable sample programs, “eScriptJMSTester.java”, demonstrates how to call System-Link MQ using JMS.

The JMS elements specific to System-Link MQ include:

- Subtree from the base LDAP URL: “ou=mq, o=psi, c=us”
- Queue Connection Factory name: “cn=XSAQueueCF”
- System-Link Request Queue name: “cn=XSAQueue”

Calling System-Link MQ via MQI

MQI is an IBM WebSphere-MQ-specific interface with support for RPG. General information on using MQI to interface with WebSphere MQ can be found at IBM’s website, and is beyond the scope of this guide. Due to the extensive nature of MQI, we recommend reviewing the downloadable sample program *USRLPI2R*. This sample program is a simple LPI implementation, and uses MQI to communicate with System-Link MQ. See Special notes on sample program USRLPI2R to view details on *USRLPI2R*.

The WebSphere MQ elements specific to System-Link MQ include:

- Queue Manager name: “COM.PSI.MAINQM”
- System-Link Request Queue name: “XSA.INPUT.QUEUE”
- Model for Dynamic Response Queue: “SYSTEM.DEFAULT.MODEL.QUEUE” (under COM.PSI.MAINQM)

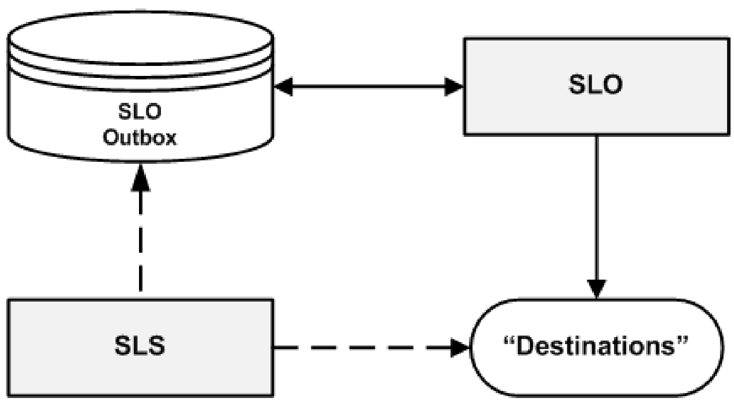
Chapter 5 System-Link Response Forwarding

5

With System-Link Response Forwarding, you can send System-Link responses to recipient programs that did not explicitly request them. (In effect, this emulates a push-based system, which means there still needs to be a request sent to System-Link that initiates this response.)

System-Link Outbox process

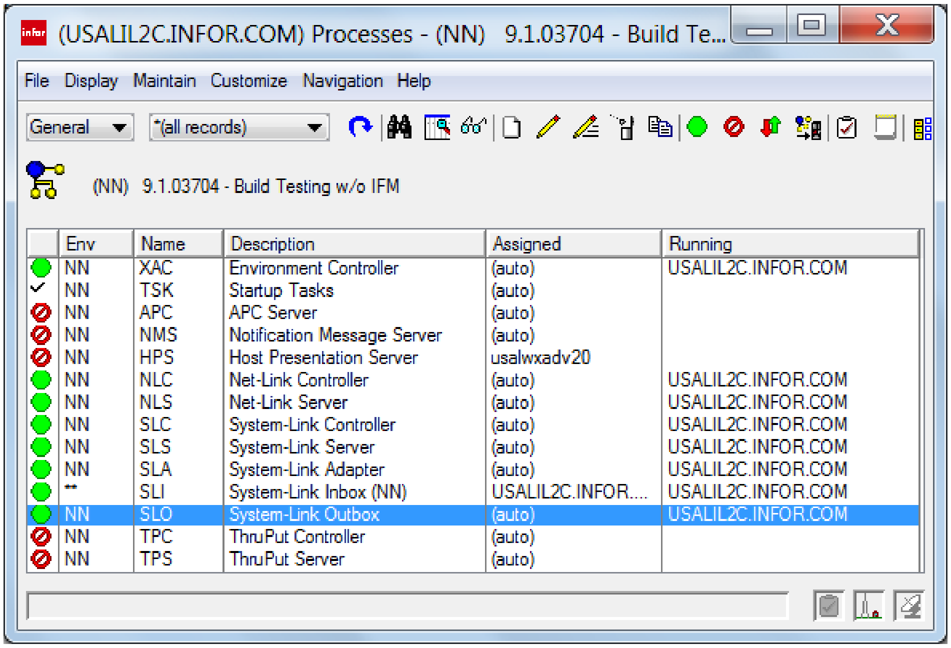
The System-Link Outbox (SLO) process, a System-Link Java server process, is used to enable resiliency and additional error detection in the System-Link Response Forwarding mechanism.



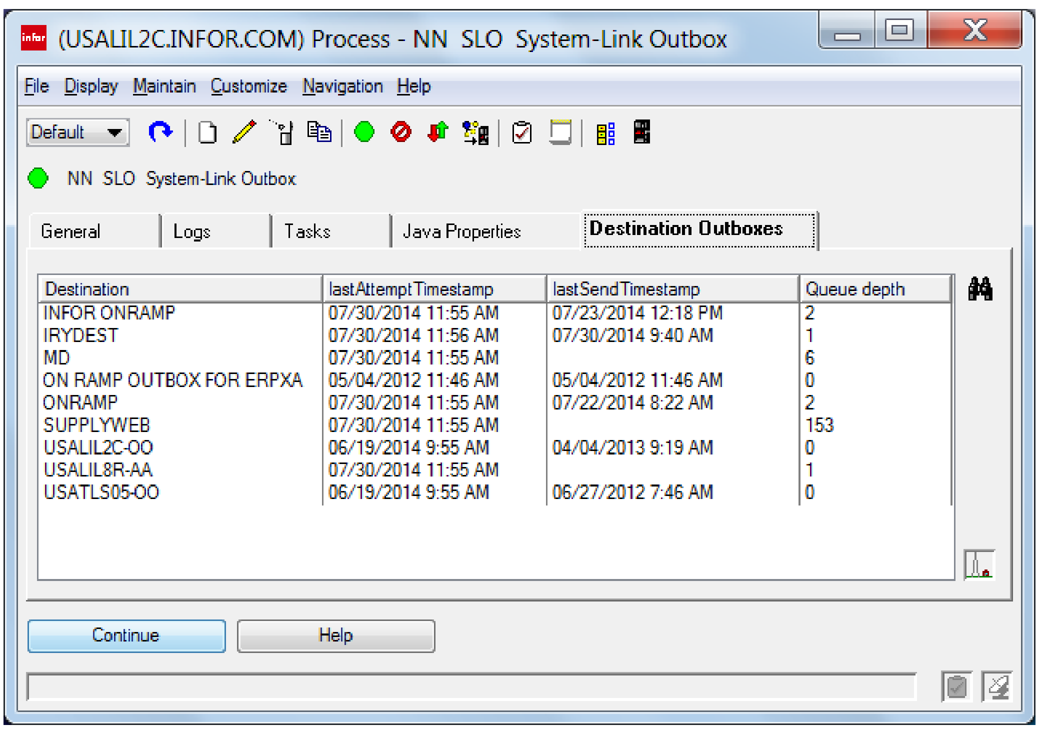
Note that SLO is only involved with outbound messages when initial attempts to forward those messages fail (that is, the receiving process is not available). If the initial forwarding action is successful, the outbound message is sent to the destination, but if the initial forwarding action fails, the failing message is queued for later retry by the SLO process.[3]

As with the other System-Link Java server processes, SLO is managed by Link-Manager.

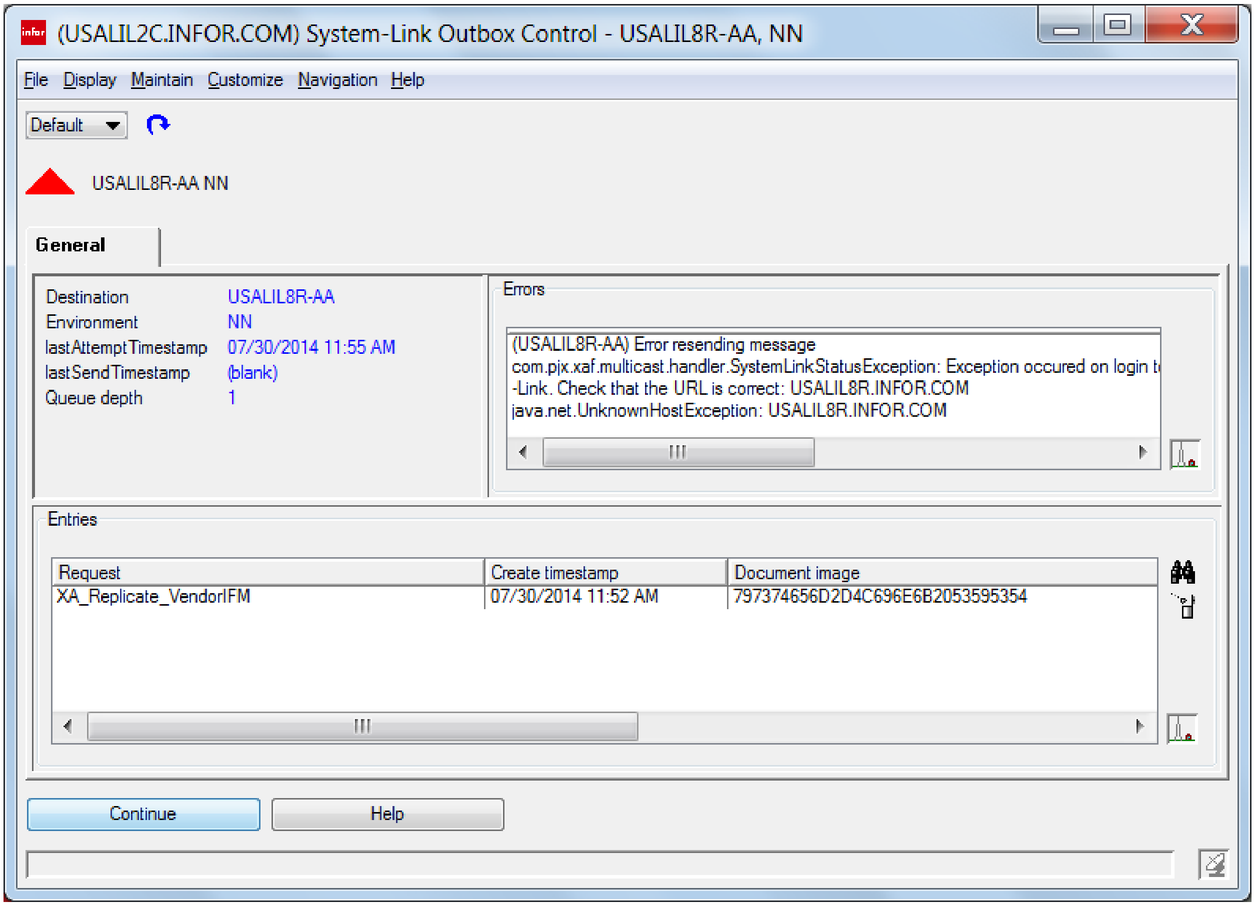
[3] The ability of System-Link to detect a failure condition will vary depending upon recipient type and nature of the failure. For example, System-Link can detect a connection failure for a Socket recipient; but, System-Link generally cannot detect semantic errors unique to a recipient's use of the forwarded response.



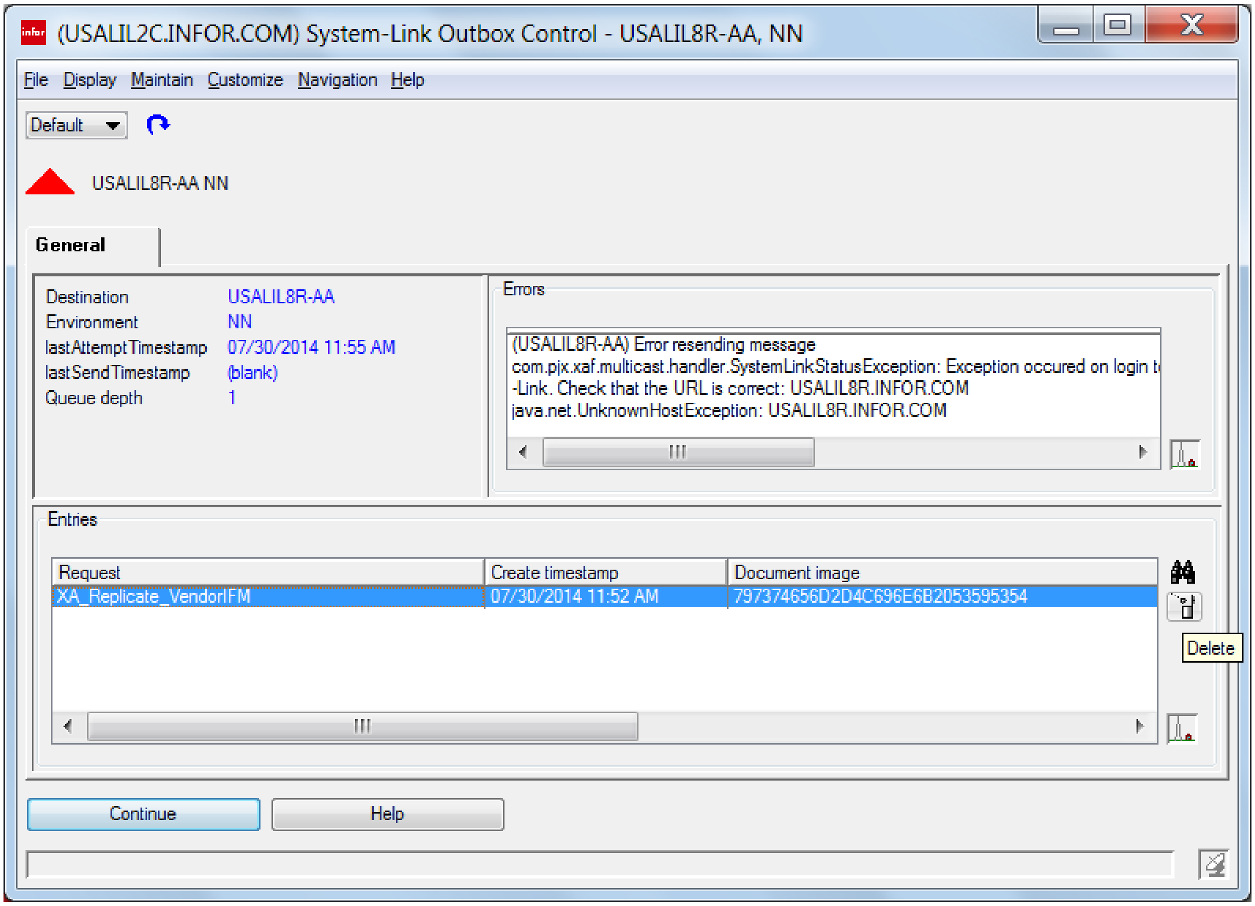
In addition to the standard Link-Manager Java server tabs, the SLO process view has a special **Destination Outboxes** tab used to manage the outbox for each System-Link Destination:



If a given Destination has send failures, then destination will show a non-zero queue length. Opening that queue will show the cause of the failure for the message at the head of the queue:



If the failure is connection-related, making the recipient program or server available will resolve the problem, and SLO will resend the message during its next retry cycle. If the failure is due to other problems, the message in error will need to be deleted from the queue before sending to that Destination can continue.



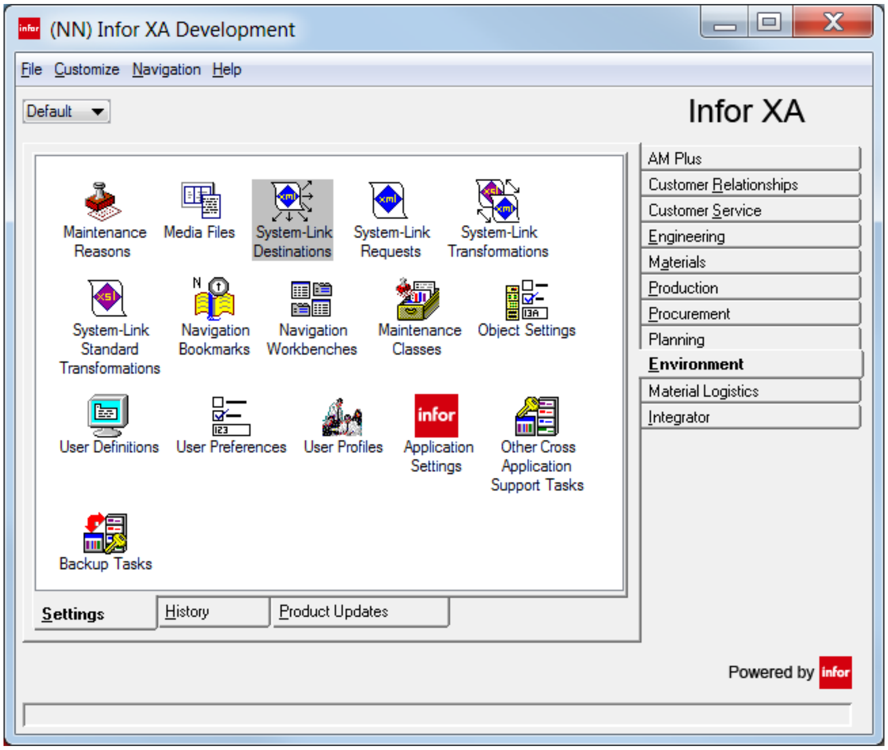
Note that within a single environment once a forwarding failure occurs for a given System-Link Destination, all subsequent forwards for that destination will queue behind the failed message. This behavior helps prevent potential problems caused by out-of-order processing of messages.

Configuration Details

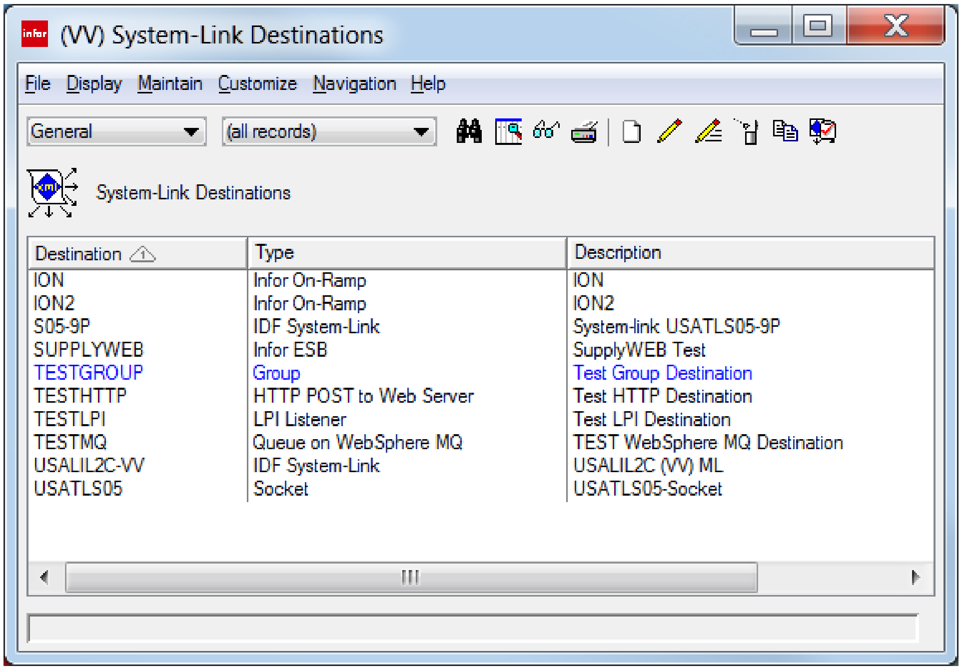
Aside from the creation of the actual recipient programs (the details of which will vary from installation to installation), you must make System-Link aware of the external recipient details, as well as identify when this recipient is to receive a copy of a given response. (The latter is specified as part of the System-Link request. See “Response Forwarding Commands” in the *IDF System-Link Reference Guide*.) The former configuration task is handled through System-Link business objects that are accessible via the Environment card on the main browser.

System-Link Destinations

For environments on which System-Link is installed, the **Settings** tab on the Environment card will contain a System-Link Destinations object:



As with other business objects, you can display a list browser and an object browser over the System-Link Destination objects.



Note: The Checkmark Action icon on the toolbar is the icon for the Test Recipient action.

At the present time, there are eight different types of destinations (at least one example of each shown above):

- Infor ESB: the recipient system is listening on the Infor Enterprise Services Bus. Infor ESB is the precursor to ION. **Note:** The Infor ESB destination type is deprecated, and is present solely to support older installations using this technology.
- Infor On-Ramp: the recipient system is listening on ION.
- IDF System-Link: the destination is another IDF System-Link environment.
- Group: the destination consists of multiple (non-group) recipients.
- HTTP Post to Web Server: the recipient program receives the response via an HTTP POST operation.
- LPI Listener: reserved for future use.
- Queue on WebSphere MQ: the recipient program is listening on a specified WebSphere MQ queue.
- Socket: the recipient program is listening on a specified socket on a specified host.

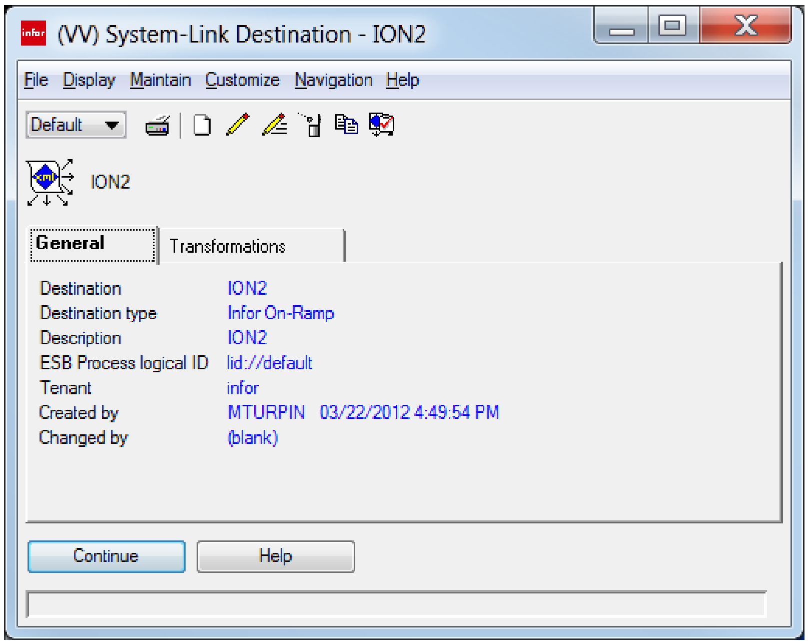
The three properties shown in the list browser above are those common to all destination types. (The purposes for the type and description properties are self-explanatory.) The value for the Destination property is used as a key in the System-Link request document to refer to a given destination. In addition, each destination type shows the System-Link Transformations available for that destination type and request. See “System-Link Transformations”.

The object browsers for each type of destination, showing the properties that are specific to each are described below.

Infor ESB

The Infor ESB destination type is deprecated, and is present solely to support older installations using this technology. Any new deployments or implementations should instead use ION and the Infor On-Ramp (ION) destination type (detailed below).

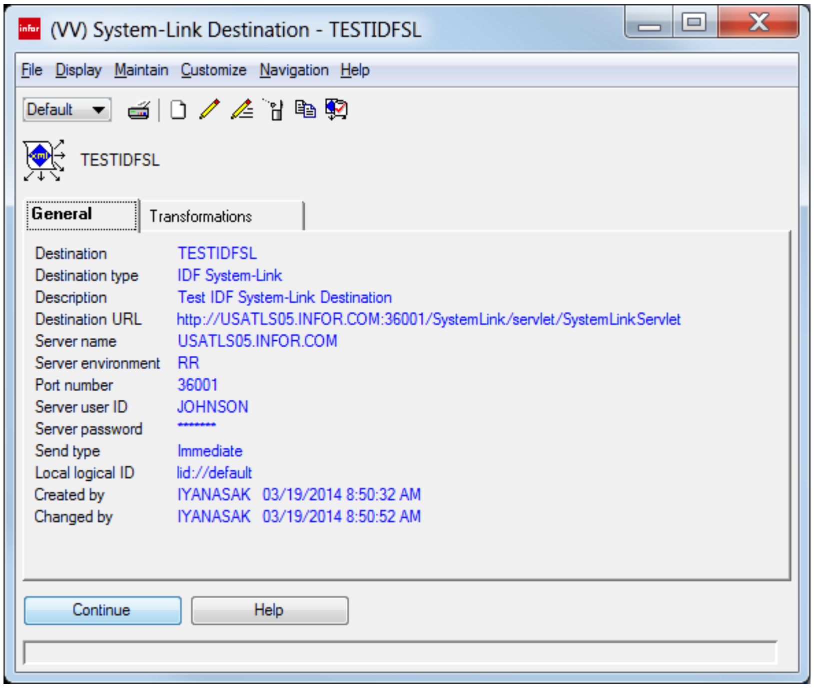
Infor On-Ramp (ION)



ESB Process Logical ID: the name of the component or subscriber that will handle the transaction transmitted through the Infor On-Ramp.

Tenant: the name of the instance of the database supporting the Infor On-Ramp.

IDF System-Link



Destination URL: the URL of the HTTP server expecting the response. This property is computed from the server name, server environment, and port numbers.

Server name: the name of the machine running the IDF recipient server.

Server environment: the identifier for the recipient IDF environment.

Port number: the listener port number configured for the IDF recipient server instance.

Server user: the user ID used to connect to the IDF recipient server (not necessarily the same as used to access the server machine).

Server password: the password associated with the user ID above.

Send type: used with the IDF Inbox (associated with the SLI process) to indicate whether responses sent to this destination should be processed immediately (Immediate), processed asynchronously (Deferred), or processed immediately, if possible (Immediate Preferred). See Send type and the IDF Inbox below for more information.

Send type and the IDF Inbox

IDF System-Link destinations have a built-in resiliency mechanism called the IDF Inbox. The IDF Inbox, managed by the SLI and SLA servers of the destination environment, allows responses forwarded to a remote System-Link server to be queued for later processing should the remote System-Link servers or environment be unavailable.

The use of the IDF Inbox is controlled in the sending environment by the value of the Send type property in that destination's definition. The Send type property has three possible values:

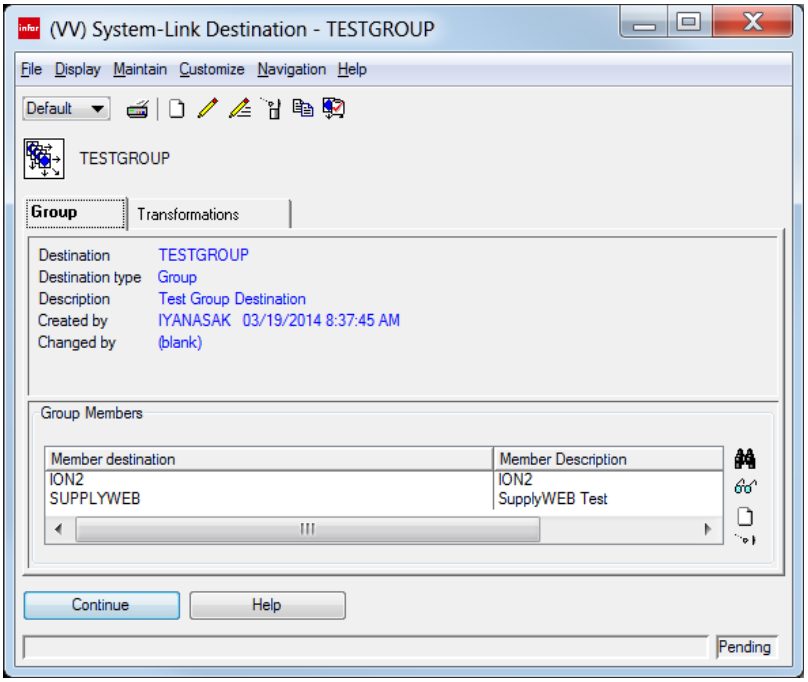
- **Immediate:** All responses forwarded to the remote destination will bypass the remote IDF Inbox, being sent directly to the remote System-Link servers for immediate processing. If System-Link is not available on the remote environment, the response forwarding attempt for this destination will fail and will not be retried. This is the default behavior for previous releases of IDF.
- **Deferred:** All responses forwarded to the remote destination will be queued in the remote IDF Inbox, to be subsequently processed by the remote System-Link servers when they are available.[4]
- **Immediate Preferred:** All responses forwarded to the remote destination will attempt to contact the remote System-Link servers directly. If these servers are not available, the responses will be queued in the remote IDF Inbox (Immediate if possible, otherwise deferred).

Note there are additional ramifications to the Send type value:

- When a forwarded document is sent to a System-Link destination configured with an ***Immediate*** Send type, the corresponding Recipient section in the response to the original System-Link request will indicate whether the send to the remote System-Link servlet was successful (including whether or not a response document was received). Therefore, when a forwarded document is queued in the remote IDF Inbox (with a Send type of ***Deferred*** or of ***Immediate Preferred*** where the remote environment or servers are not available), the corresponding Recipient section only indicates if the forwarded document has been successfully queued. There is no subsequent indication sent to the System-Link request originator when that queued document is subsequently processed.
- If it is important that forwarded responses are processed on the remote system in the order that the requests were submitted, then one should choose ***Immediate*** as the Send type, and then wait for each response document to indicate that the forwarding action was successful. For ***Deferred***, each request will be processed in the order that it was queued, but errors encountered when processing a document will not prevent subsequent documents in the queue from being processed. For ***Immediate Preferred***, a sudden temporary unavailability of the remote System-Link servers could cause some documents to be queued while subsequent ones are processed immediately, thereby potentially altering the processing order.

[4] Note that deferred responses, from either ***Deferred*** or ***Immediate Deferred*** Send type settings, will only be queued in an environment's IDF Inbox if that environment's SLI process is running. Each environment's SLI process is started and stopped with the Global environment; SLI is not affected by starting and stopping its associated environment within Link-Manager

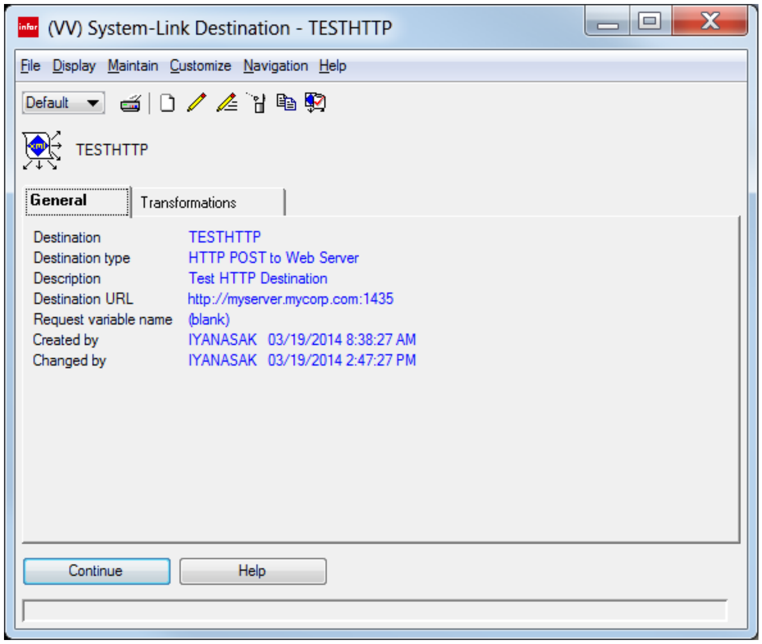
Group



The Group destination type contains an editable list of member destinations. These members can be of any destination type except for Group.

HTTP Post to Web Server

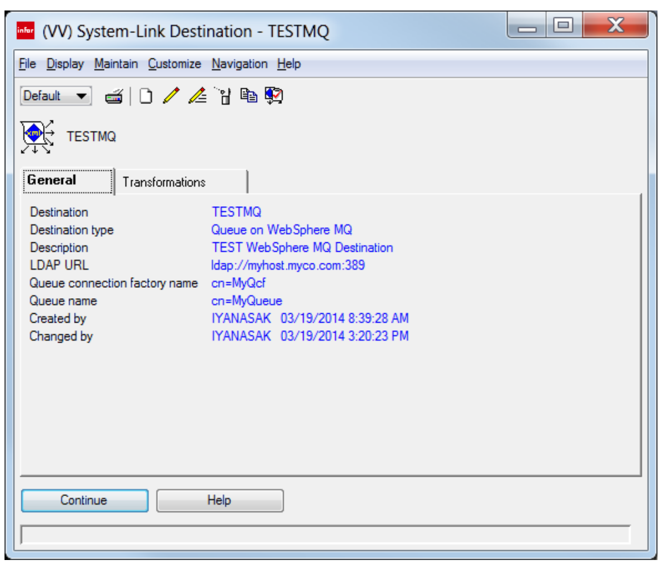
Note an HTTP Post to Web Server destination type represents a connection to a generic servlet on a remote machine. For connections to a System-Link servlet on a remote machine use the **IDF System-Link"** destination type.



Destination URL: the URL of the HTTP server expecting the response.

Request variable name: (optional) if the server is expecting the response to be the value for a variable in the POST body (as opposed to simply being the body itself), this variable name is specified here.

Queue on WebSphere MQ

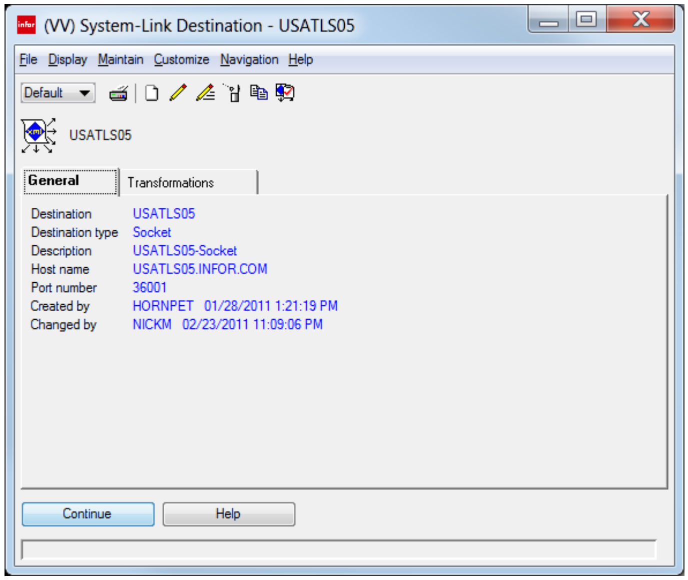


LDAP URL: the URL of the LDAP server on which the queue connection factory and queue are bound.

Queue connection factory name: name of the queue connection factory.

Queue name: name of the queue on which the response should be placed.

Socket



Host name: name of the host machine of the recipient.

Port number: number of the port on which the recipient is listening.

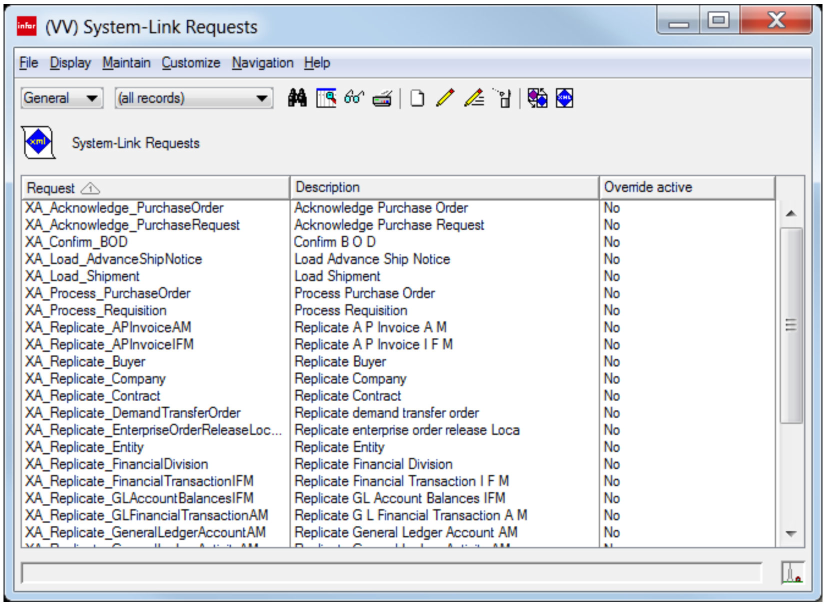
System-Link Requests

For environments on which System-Link is installed, the **Settings** tab on the Environment card also contains a System-Link Requests object. A System-Link request provides information required to form requests to send information to another system. Each System-Link request contains descriptive information about the request, control information for how to process the request, audit controls to log the request, and the original and overridden versions of the request XML.

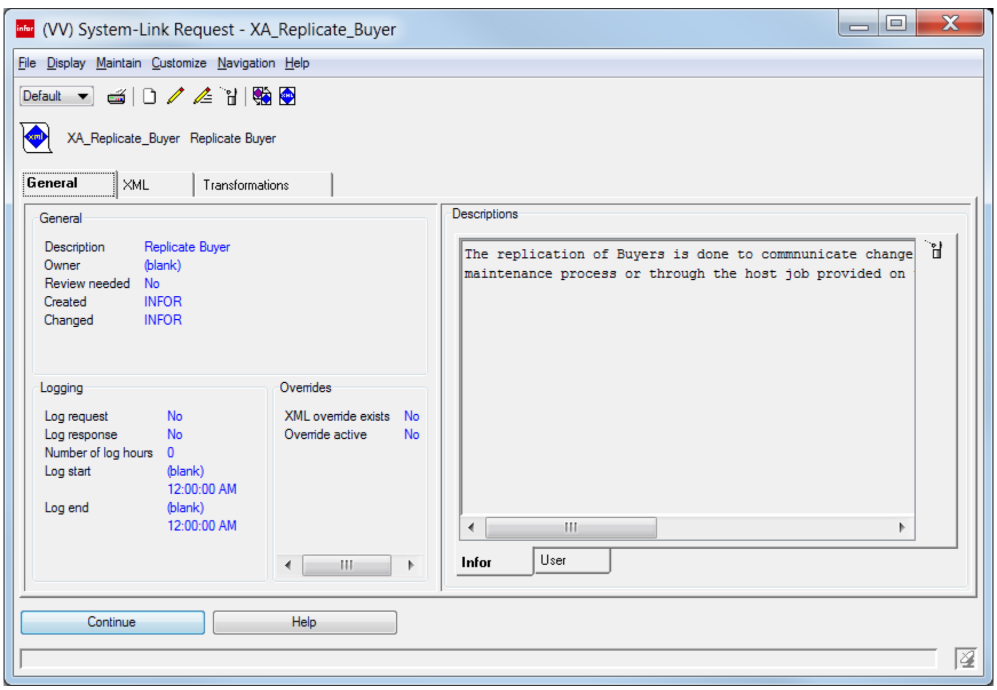
Infor provides a default request for each transaction supported by a BOD. Users can use or modify the XA, LX, or System21 requests or create additional requests. The Request ID for Infor-provided requests begins with “XA_”, "LX_", or "S21_", and contains a verb and noun indicating the type of request.

The list browser over the System-Link Requests object shows the request ID, description, and whether the body of the XML for the request has been overridden.

Request IDs for user-defined requests cannot begin with XA, LX, or S21. IDF reserves the request ID of *INBOUND for a predefined request.



The object browser shows properties specific to a selected request. The General card contains processing information for the request and a list of request transformations defined for the request. This card combines information from files SLRHDR, SLRTXT, SLRUXT, and SLRTRF.



Owner: the ID of the user who has responsibility for the request. The owner cannot be changed after the request is created. IDF sets the owner for all requests provided by Infor to (blank).

Review needed: Yes indicates that the request requires review. IDF sets this attribute to Yes automatically for any new versions of an Infor-supplied request.

Log request: indicates whether transmission of the request is logged in the Transaction Status object.

Log response: indicates whether the response to the request is logged in the Transaction Status object.

Number of log hours: the number of hours for the duration of logging. This time, combined with the Log start date and time, can be used to determine the log end date and time automatically.

Log start: the day and time for the log activity to begin.

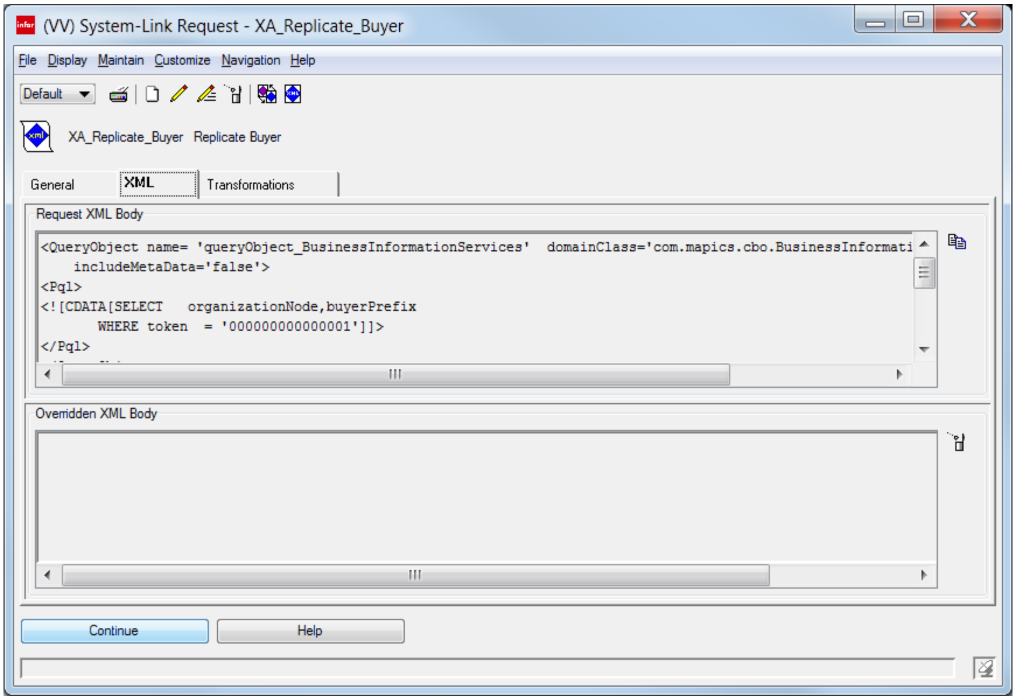
Log end: the day and time for the log activity to end. If specified, the Number of log hours can be used with the Log start values to calculate the Log end day and time automatically.

XML override exists: Yes indicates overridden XML content on the XML tab is active or ready for use. The default value is No. This attribute is available if user-entered XML content exists for this request.

Override active: Yes indicates the overridden XML is used for the body of the request instead of the original XML. IDF automatically sets this attribute to Yes if override XML is entered for the request.

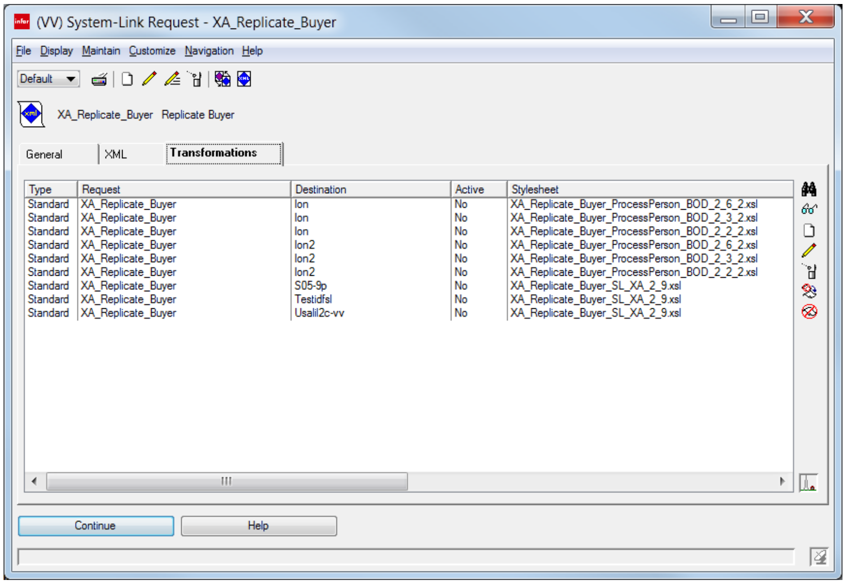
The Descriptions section contains the text from the SLRTXT and SLRUTX files.

The XML card shows the request XML provided by Infor from the XML Requests file (SLRXML) and an area for entry of custom XML that overrides the predefined XML for the body of the request. Any custom XML entered is stored in the User Overridden XML Requests file (SLSUXM). For XML requests provided by Infor, the Request XML Body can be overridden by XML entered in the Overridden XML Body. The overridden Request XML Body does not replace the original Request XML Body but is used in the request transaction if the XML override exists and Override active attributes are set to Yes.



IDF provides the XML for the login, header, and footer sections of the System-Link request and for testing the request. The XML in these sections of the request cannot be changed. Users can create System-Link requests, but the XML for the login, header, footer, and test sections defaults to the Infor-provided XML. Users provide the XML for the body of the non-IDF requests.

The Transformations card lists all System-Link request transformations that apply to this request. The related transformations come from the SLRTRF file.



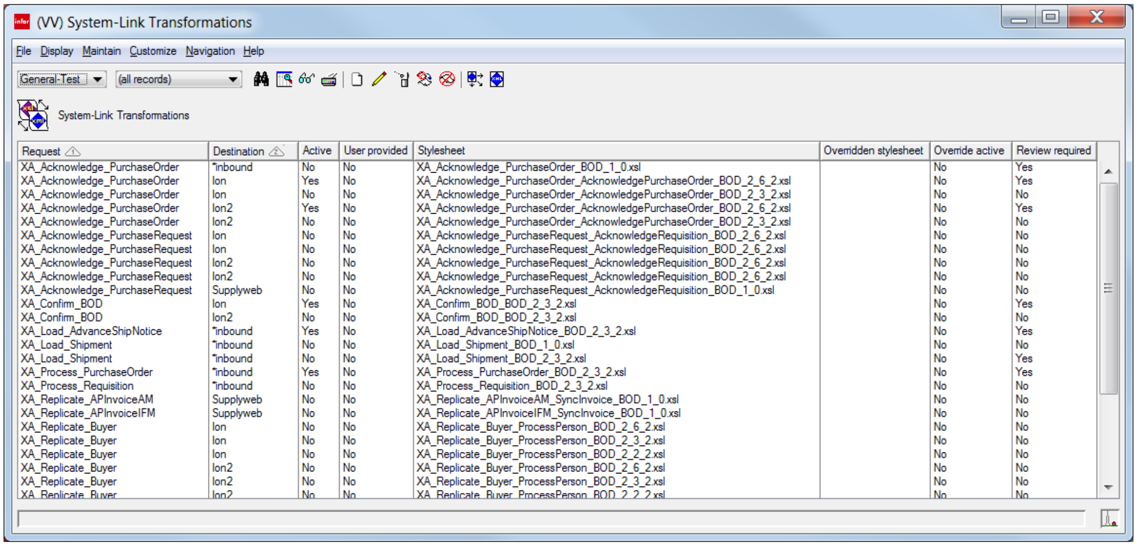
System-Link Transformations

Most System-Link Destinations will not accept a raw System-Link response document as input. Instead, the response document must first be converted into a document of a different format (for example, an OAGIS BOD). System-Link Transformation instances, in conjunction with their associated stylesheets, tell System-Link how to perform this conversion.

Each row in the list browser of the System-Link Transformations application object represents a single instance of:

- Any transformation explicitly assigned to a single System-Link Destination, or
- A Standard Transformation implicitly associated with a single System-Link Destination of a matching type

Standard Transformations (and their stylesheets) are provided by Infor, along with their matching System-Link Requests. Standard Transformations appearing in this list browser will never have overriding stylesheets, nor will they ever be active. Infor currently provides Standard Transformations only for Infor On-Ramp, Infor ESB, and IDF System-Link destination types.



Request: the name of the System-Link Request instance associated with the transformation.

Destination: the System-Link Destination explicitly assigned to or implicitly associated with the transformation. Transformations applying to inbound documents will show "*inbound" in this column.

Active: is the transformation instance active?

User provided: is this a User Transformation? User Transformations have no Infor-provided stylesheets, only having override stylesheets defined. This distinguishes them from explicitly-assigned transformations that are based upon (Infor-provided) Standard Transformations, but which include active override stylesheets.

Stylesheet: the Infor-provided stylesheet associated with the transformation. For User Transformations, this field will be blank.

Override stylesheet: any user-provided stylesheet that (if active) will be used instead of the listed Infor-provided stylesheet. For User Transformations, an override stylesheet must always be specified.

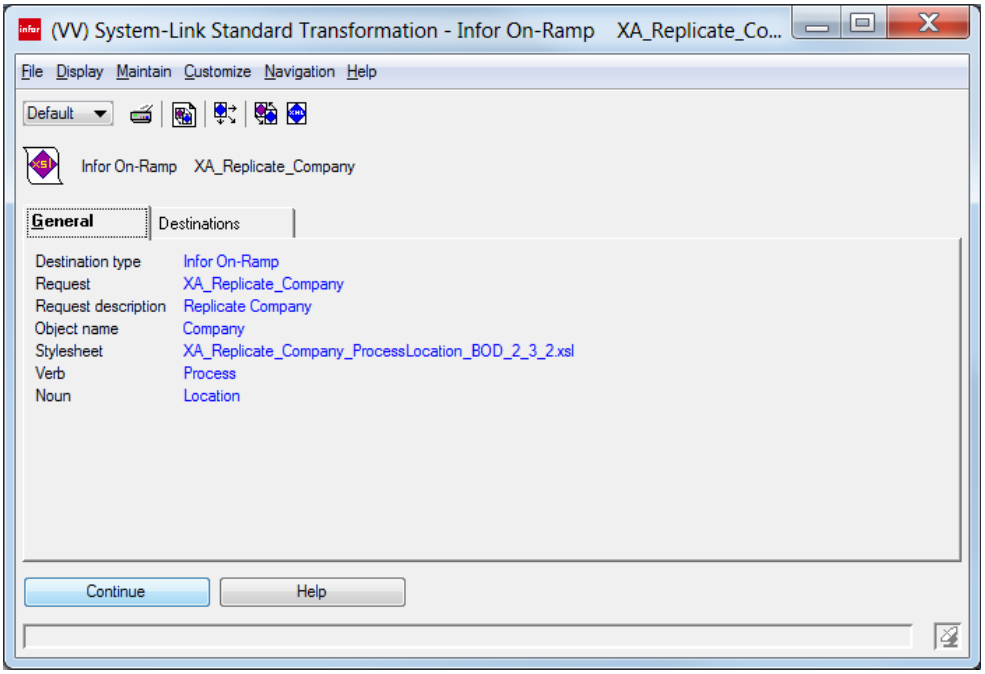
Override active: will the named override stylesheet be used instead of the listed Infor-provided stylesheet?

Review required: does this transformation require review? IDF sets this attribute to *Yes* automatically for any new versions of an Infor-supplied transformation.

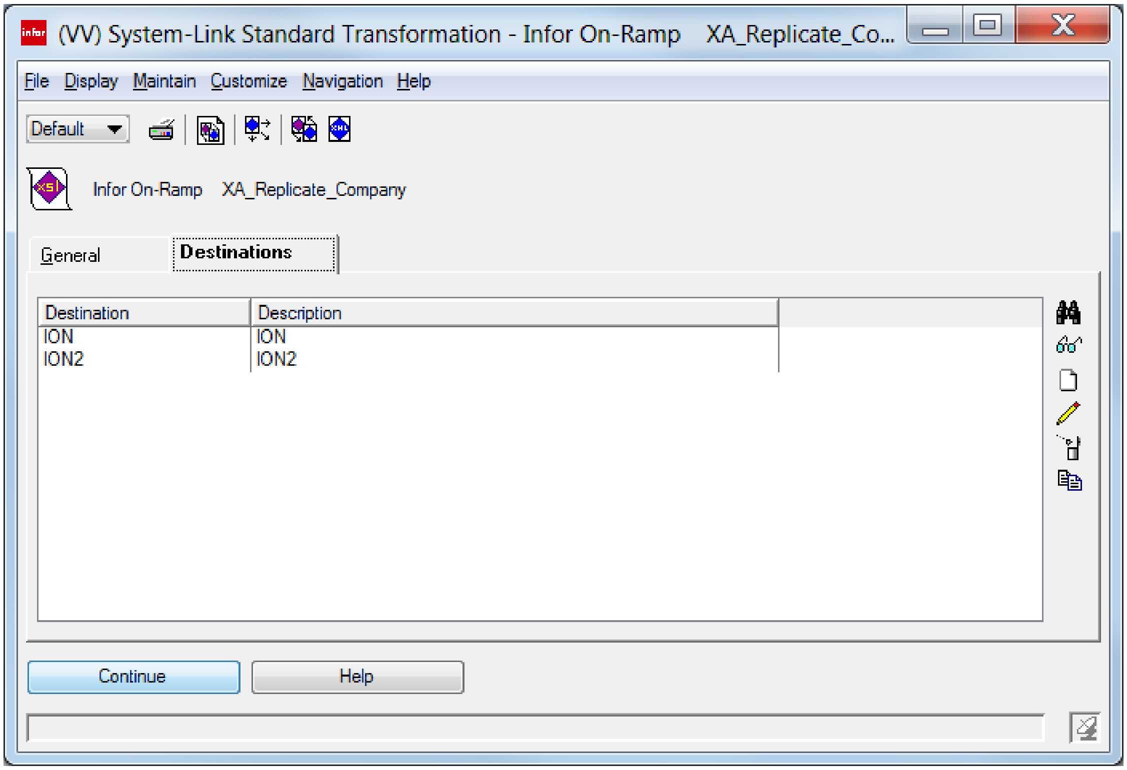
When any row in the Transformations list browser is selected, Power-Link will open one of two transformation object browsers depending upon that type of transformation.

Standard Transformation Browser

If the transformation instance represents a Standard Transformation, then Power-Link will launch the Standard Transformation object browser.



The properties shown on the General card are the default ones for the Standard Transformation. They include the matching destination type, the associated System-Link Request details (name, description, object name), the Infor-provided stylesheet, and the verb and the noun for the associated OAGIS BOD.



The Destinations list card displays all of the System-Link Destinations that are implicitly associated with this Standard Transformation. By definition, this will include the Destination listed for the row in the Transformations list browser.

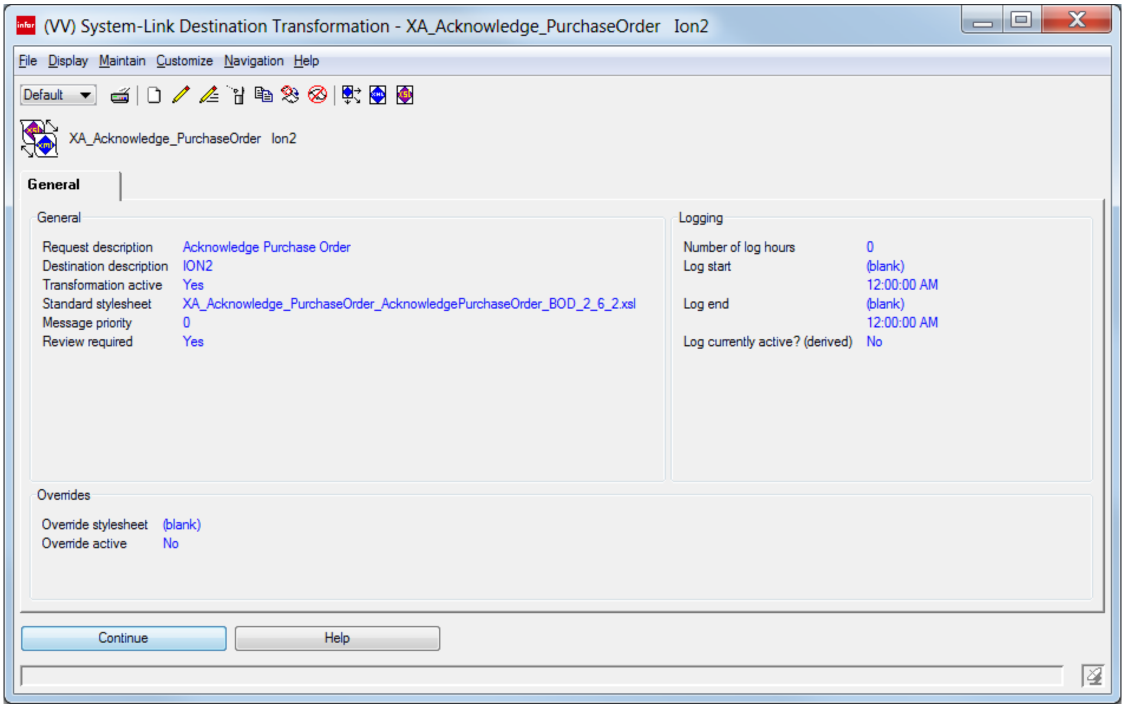
Note: As mentioned above, any Standard Transformation appearing in the Transformations list will never be active[5].

[5] If the Standard Transformation entry listed in the Transformations list browser is activated, from that point forward the Destination Transformation object browser will be displayed for that entry, even if it is subsequently deactivated. The only way to revert the entry back to a Standard Transformation is to delete the entry, which will also deactivate it as a consequence.
This behavior is designed to persist override or logging settings that the admin may have set on that particular transformation instance.

Destination Transformation Browser

If the transformation instance has been user created or modified in any way, including activation, then Power-Link will launch the Destination Transformation object browser.

Note: Unlike Standard Transformations, a Destination Transformation instance always represents a one-to-one association between a transformation and a specific destination.



General

Request description: the detailed description of the associated System-Link Request.

Destination description: the name of the single System-Link Destination to which this transformation applies.

Transformation active: indicates that the transformation is active.

Standard stylesheet: if this transformation is based on a Standard Transformation, this lists the associated Infor-provided stylesheet.

Message priority: reserved for future use.

Review required: indicates if this transformation requires review. IDF indicates that review is required automatically for any new versions of an Infor-supplied transformation.

Overrides

Override stylesheet: the name of a user-provided stylesheet to be used in place of the standard stylesheet (if active).

Override active: indicates if the active stylesheet should be overridden.

Logging

Number of log hours: the number of hours for the duration of logging. This time, combined with the Log start date and time, can be used to determine the log end date and time automatically.

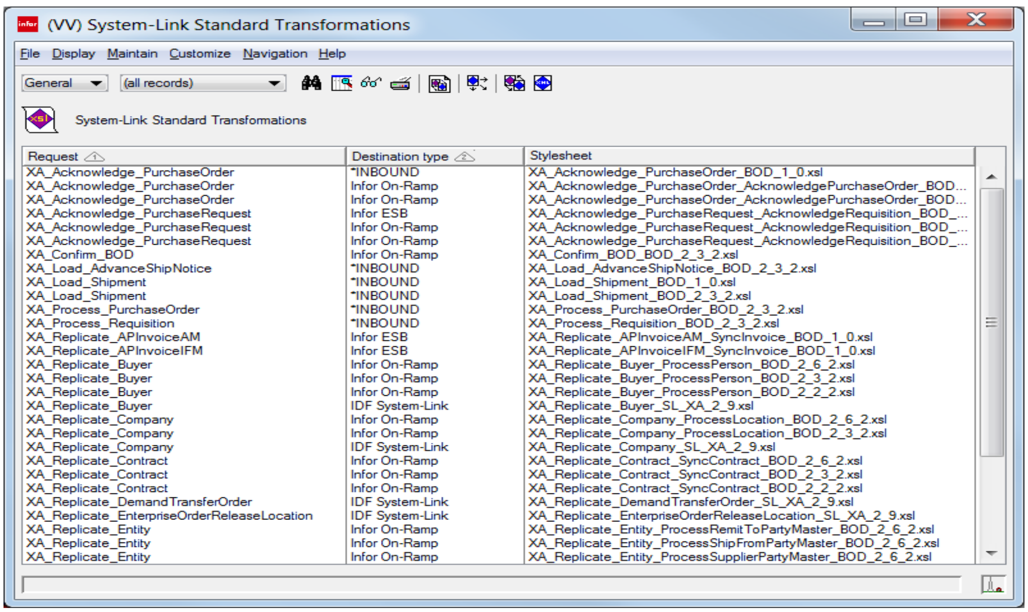
Log start: the day and time for the log activity to begin.

Log end: the day and time for the log activity to end. If specified, the Number of log hours can be used with the Log start values to calculate the Log end day and time automatically.

Log currently active?: based upon the above settings, is logging currently active for this transformation.

System-Link Standard Transformations

The final System-Link object on the Settings tab on the Environment card is System-Link Standard Transformations. The associated list browser shows a list of all Infor-provided transformations.



Displaying any of the transformations in this list will show the transformation details and destinations in the Standard Transformation object browser discussed previously.

Transformation Mechanism

Transformations are performed once per stylesheet against the generated System-Link response. System-Link processes the transformation a single time for each stylesheet because the transformed document could be sent to multiple destinations. The document sent to each destination should contain information specific to that destination. The transformed document varies for each destination due to substitution variables that differ among destinations. These variables are primarily, but not exclusively, obtained from the settings specified for the destination. The variable values can be inserted into the transformed document by specifying the substitution variable in the stylesheet and are delimited by the “%” character. Not all destination types use substitution variables. The different requirements for substitution variables used with different destination types are described in the following sections. For example, the code below specifies the Session handle value for a System-Link destination of type HTTP.

```
<Request sessionHandle=”%SESSIONHANDLE%” workHandle+”*new” broker=”EJB”
maxIdle=”1000”>
```

Additionally, some of these substitution variables might require data from the generated document (for example, the BOD ID on Infor Bus 2.0 and Infor On-Ramp). To provide this data, reference values should be added in the stylesheet. The reference values can be added by specifying a comment in the stylesheet that contains the text TX_REF:{Reference}={Value}.

Note: Not all destination types rely on this mechanism. The following example specifies the ReferenceID value used when building the BOD_ID substitution variable for Infor Bus 2.0 destinations. The entire example should be entered on a single line, but the line is wrapped here due to formatting limitations in this document.

```
<xsl:comment>TX_REF:ReferenceID=<xsl:value-of select=”System-\Link/Response/Query
ObjectResponse/DomainEntity/Key/Property[@path=’order’]>/Value”/></xsl:comment>
```

System-Link HTTP post

A destination of this type performs the login and session maintenance automatically using the values specified in the System-Link Destination details. When creating the stylesheet for this type of destination, it is necessary to assume that the login has been performed. Therefore, all elements that require the sessionHandle attribute should specify this value using the SESSIONHANDLE substitution variable.

Substitution variables

SESSIONHANDLE	The session handle obtained from the login. Use this value wherever the sessionHandle attribute is required.

Transformation references

There are no transformation references.

Infor Bus 2.0

Connection to the Bus will be performed automatically. The substitution variables should only be used inside the ApplicationArea section because only this area can differ between destinations. These details will be obtained from the Bus properties table and from the destination details.

Substitution variables

TO_LOGICAL_ID	The logical ID of the destination. For Process or Get verbs, this value will be specified in the destination. For all other verbs, this value will be lid://default.
FROM_LOGICAL_ID	The logical ID of the sender. This value will be obtained from the Bus properties.
CREATION_DATE_TIME	The current time as a formatted timestamp in GMT.
BOD_ID	The generated BOD ID has a pre-generated format which includes the value obtained by referring to the transformation reference (ReferenceID).
REFERENCE_ID	Currently the same value as BOD_ID.

Transformation references

ReferenceID	The format of the reference ID used to generate the BOD ID is related to the key of the business object. This transformation reference is used to specify the value to be used.

Example

The following example is a fragment of the stylesheet used to send a Purchase Order with the Process verb. The ApplicationArea of the example should be used for most Infor BOD level 1.0 transformations.

```
<ProcessPurchaseOrder ... >
  <xsl:comment>TX_REF:ReferenceID=<xsl:value-of select="System-
  Link/Response/QueryObjectResponse/DomainEntity/Key/Property[@path='order']/Value"/
  ></xsl:comment>
  <ApplicationArea>
        <Sender>
              <LogicalID>%FROM_LOGICAL_ID%</LogicalID>
              <ReferenceID>%REFERENCE_ID%</ReferenceID>
        </Sender>
        <CreationDateTime>%CREATION_DATE_TIME%</CreationDateTime>
        <BODID>%BOD_ID%</BODID>
  </ApplicationArea>
```

Infor On-Ramp

The substitution variables should only be used inside the ApplicationArea section and verb sections because only these areas can differ between destinations. These details will be obtained from the Bus properties table and from the destination details.

Substitution variables

TO_LOGICAL_ID	The logical ID of the destination. For Process or Get verbs, this value will be specified in the destination. For all other verbs, this value will be lid://default.
FROM_LOGICAL_ID	The logical ID of the sender. This value will be obtained from the Bus properties.
CREATION_DATE_TIME	The current time as a formatted timestamp in GMT.
BOD_ID	The generated BOD ID has a pre-generated format which includes the value obtained by referring to the transformation reference (ReferenceID).
MESSAGE_ID	A system-generated identifier for the BOD that is added to the message headers. This value currently is not used in the ApplicationArea.
TENANT_ID	The identifier for the tenant. Currently, this variable has a fixed value of “infor.”
ORGANIZATION_NODE	The identifier for the organization. This value is obtained directly from the transformation references.

Transformation references

ReferenceID	The format of the reference ID used to generate the BOD ID is related to the key of the business object. This transformation reference is used to specify the value to be used.
AccountingEntityID	The accounting entity ID is optional. It is not required when the object does not relate to a single accounting entity, but spans multiple entities. Used when building the BOD ID.
LocationID	The location ID is optional. It is not required when the object does not relate to a single location. Used when building the BOD ID.
Revision	The revision of the business object is optional. Used when building the BOD ID.

Example

The following example is a fragment of the stylesheet used to send a buyer in the form of a Person, using the Sync noun. Note that none of the optional transformation references are defined.

```
<SyncPerson …>
  <xsl:comment>TX_REF:ReferenceID=<xsl:value-of select="System-
  Link/Response/QueryObjectResponse/DomainEntity/Key/Property[@path='buyer']/Value
  "/></xsl:comment>
```

```
    <ApplicationArea>
        <Sender>
            <LogicalID>%FROM_LOGICAL_ID%</LogicalID>
            <ComponentID>ERP</ComponentID>
                </Sender>
                <CreationDateTime>%CREATION_DATE_TIME%</CreationDateTime>
                <BODID>%BOD_ID%</BODID>
            </ApplicationArea>
            <DataArea>
                <Sync>
                    <TenantID>%TENANT_ID%</TenantID>
                    <AccountingEntityID>%ACCOUNTING_ENTITY_ID%</Accounting
                    EntityID>
                    <ActionCriteria>
                        <ActionExpression><xsl:attribute name="actionCode">
                            <xsl:call-template
                            name="ActionCode"/></xsl:attribute>
                        </ActionExpression>
                    </ActionCriteria>
                </Sync>
    …
</SyncPerson>
```

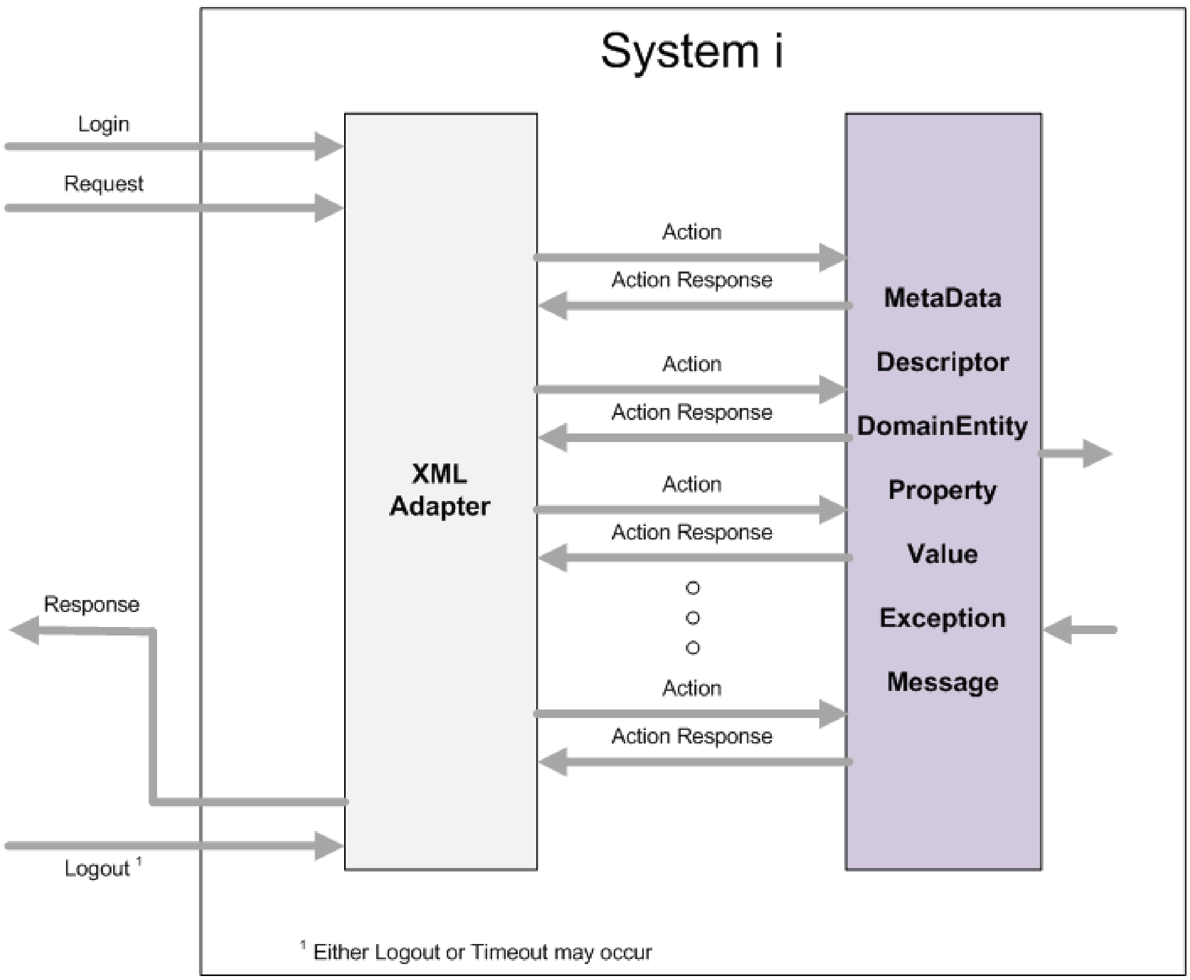
The Test Destination Action

Once the configuration data of a recipient is entered, the Test Destination action provides a way to validate that the connection can be made. Note that this validation will require the associated recipient program to be present and active (and ready to catch the test response). This implies that the supporting systems, such as WebSphere MQ for Queue on WebSphere MQ destinations and the remote web server for HTTP Post to Web Server destinations, should be installed and running as well.

Chapter 6 The System-Link Protocol in Detail

6

IDF System-Link uses an XML-based request/response protocol that is transported via one of several communication links (for example, an HTTP POST request (with MIME type text/xml), a WebSphere MQ message, an RPG program call, an ION message, etc.).



The base unit of a dialog with IDF System-Link is the session. A session consists of a login action, one or more request actions, and an explicit logout action or an implicit timeout. The login determines user rights via IDF security, and specifies other session parameters (such as, IDF environment, national language, etc.). Logout (via a logout action or a timeout), as usual, simply terminates the session. The request action(s) contain all of the real work of a session that is performed in the context of a work area. The work area contains all temporary and non-persisted data. A given session can have one or more work areas, but a single work area cannot span sessions.

A key feature of both sessions and work areas is their timeout mechanism. The session timeout is specified at login, whereas the work area has its timeout specified at the beginning of the request action. This feature allows both sessions and work areas to persist beyond the processing of a single System-Link request script (or to idle), which has these consequences:

- The login, request, and logout actions can be split across multiple request/response pairs. This allows you to avoid incurring the considerable login overhead more than once per dialog. In addition, this allows System-Link to be installed in such a way that the login/logout and the request operations are sent to different ports such as allowing SSL for login/logout while handling the normal request data in the clear.
- Matching the session and work area idle times more closely to the expected usage pattern allows session and work area reuse while minimizing server overhead, license usage, and security window size.
- Once a session expires all of its work areas will also expire. A work area cannot exist outside of its enclosing session.
- Since an IDF login may require up to several seconds to complete, the performance benefit from reusing one login session (as opposed to creating a new login session for every request) can be substantial.

Note that although work areas can be shared between requests as well, there is marginal performance and overhead benefit to be gained by doing so. Unlike sessions, work areas can be invalidated for several reasons (for example, transaction failures), and handling such conditions requires additional logic in the sending system. In general, unless a request has retained results and named objects (detailed later) that will be used in a subsequent request; work area reuse is not advised.

Inside a given request action, the data query and maintenance requests (actions) are specified. Query action elements can be used to request a list of business domain objects, a single domain object, or a single domain object followed by one or more lists of related domain objects. Maintenance action elements can be used to create, copy, update, or delete a domain object. Any given request can contain one or more actions. Each response to a given action will be associated with the action by name, and will contain the results of the action or one or more exception messages detailing the errors that occurred. (More information about the format and types of exception messages can be found under “Exception element” in the *System-Link Reference Guide*.)

Queries can be specified using one of two methods: PQL or custom definitions. The Paragon Query Language(PQL) shares much of its syntax with SQL, and therefore should be fairly easy to understand for those already familiar with SQL. When viewing PQL, the SQL-literate person will often run across the following notable difference (example: a SELECT clause fragment for the domain class “PurchaseOrder”):

“SELECT order, relatedBuyer.buyerName, relatedVendor.vendorName1 … ”

In the above, order specifies that the PurchaseOrder property order should be displayed. Likewise, relatedBuyer.buyerName and relatedVendor.vendorName1 specify that buyerName and vendorName1 be displayed; although these two properties are not present on PurchaseOrder. Instead, they <u>are obtained via known relationships with that given PurchaseOrder</u>. In this case, buyerName would be obtained from the related Buyer, and vendorName1 would be obtained from the related Vendor.

An alternative way of specifying a query can be using custom definitions. System-Link can determine which data to retrieve based on a view name rather than a SELECT clause in PQL. Similarly, a subset name replaces the WHERE clause and a sort name replaces the ORDER BY clause. The advantage of using custom definitions instead of PQL is that the results of the XML request can be controlled by simply creating and maintaining the referenced custom definitions.

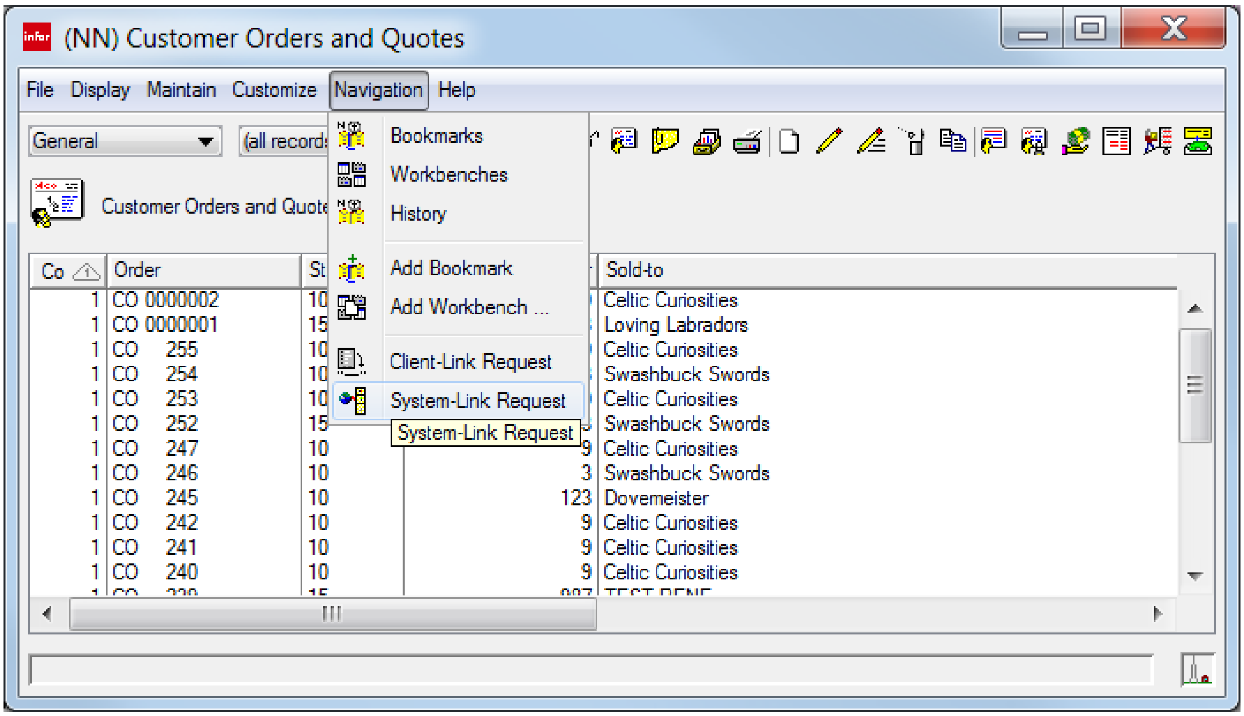
Global Restrictions

A named action (for example, QueryObject, FetchNext, Update) cannot have the same name as any other named action in the same System-Link document. Therefore, names can be reused across documents for a given System-Link session and work area. **Note:** Reuse of names does limit the use of named objects. Named objects are detailed later in this guide.)

Chapter 7 Building a System-Link request using Power-Link

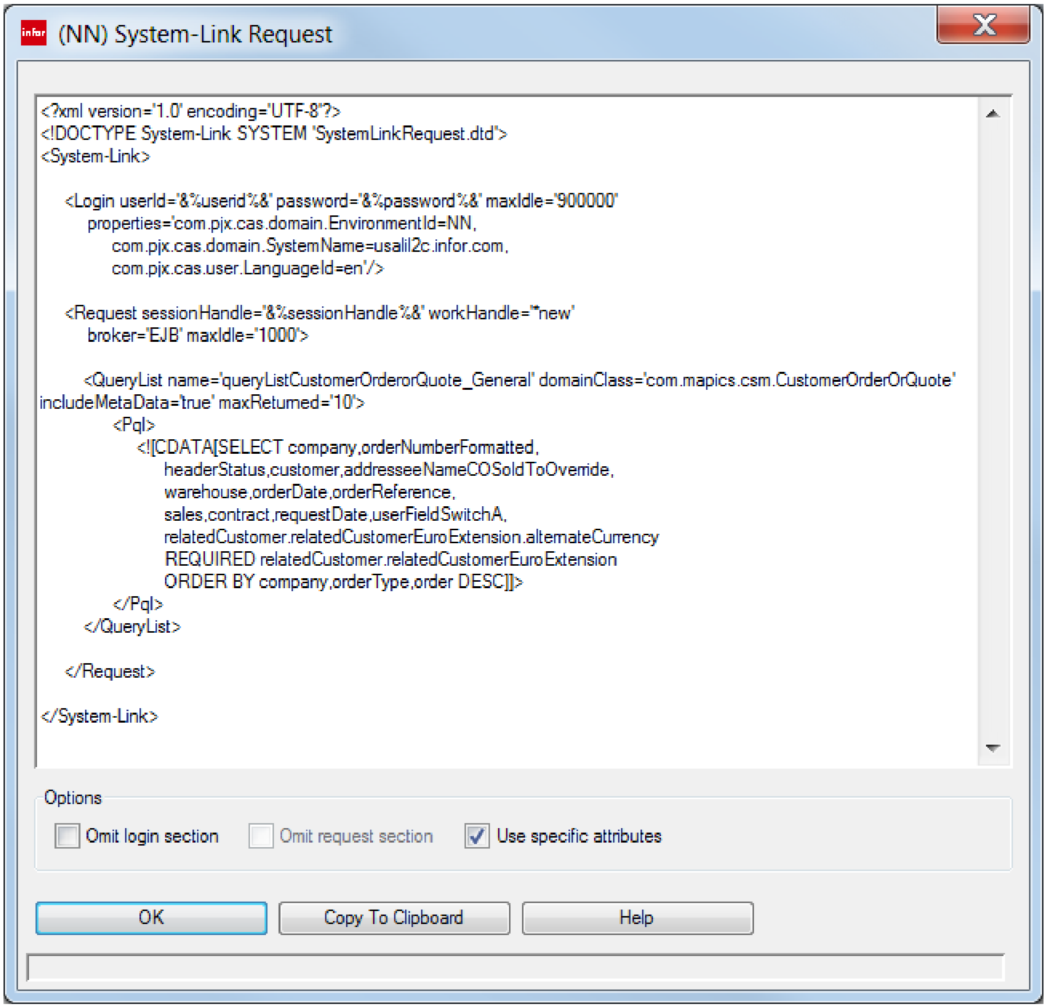
7

The process of creating a System-Link request by hand can be both tedious and subject to errors. To assist you, Power-Link provides tools to build the System-Link request from the menu bar.



On a list window or card file, select the **System-Link Request** option from the **Navigation** menu.

The System-Link Request window opens.

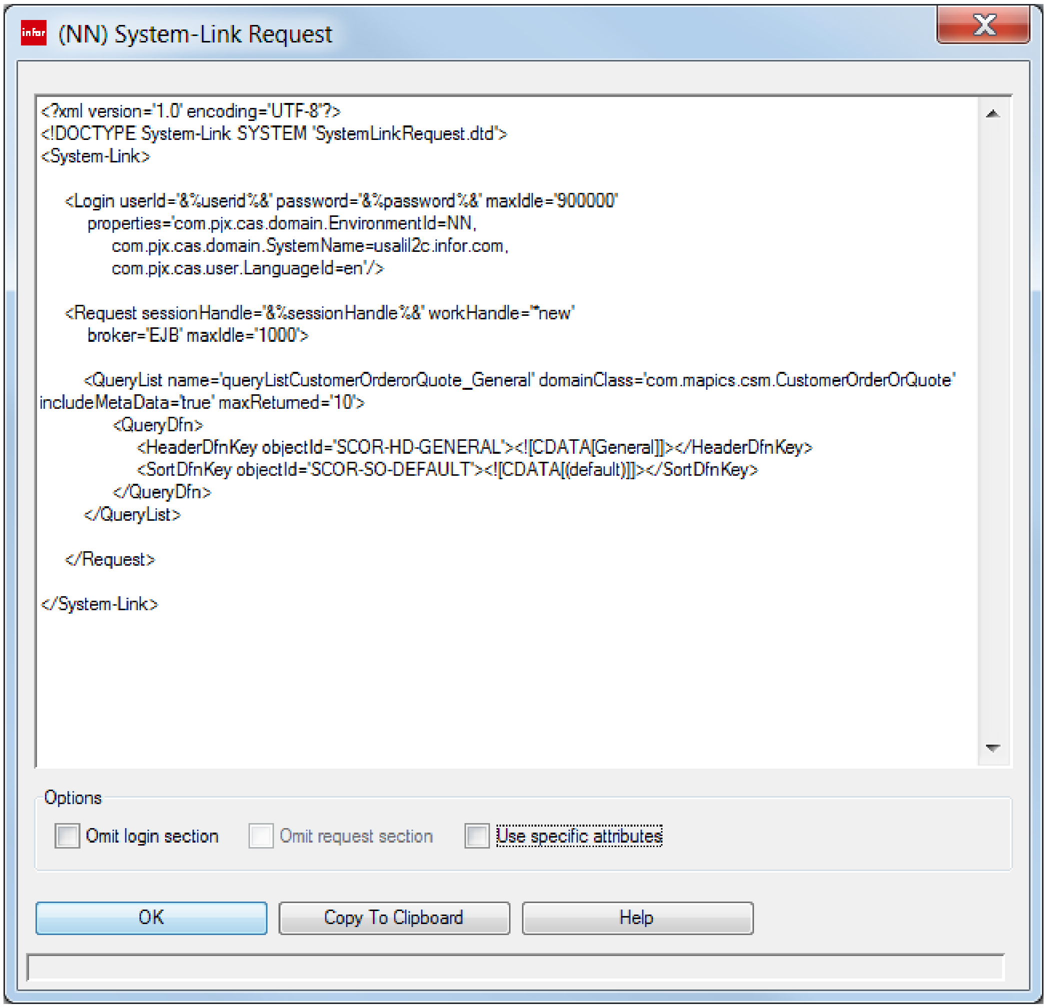


This window shows the System-Link request using the data from the list window or card file. When you ask for the System-Link request for a view, the attributes included in the request are only those listed on the banner or as columns in the view.

Similarly, the System-Link request for a card file includes all the attributes on all the cards in the file, plus any in the banner. If there are customizable list cards in the card file, the request includes the data on those list cards as well.

System-Link allows you to specify the attributes discretely or by the custom definition. In the example above, the attributes for a list of items are identified in the PQL: Item, Description, Item class, etc. (The unusual capitalization and lack of spaces in the attribute names is necessary.)

If you clear the **Use specific attributes** option, the System-Link request is altered to reference the associated custom definitions:



The System-Link request is shown in sections. The login section includes the user ID, password, environment, and other login information. The user ID and password are not displayed. Instead, you see the words "*userid*" and "*password*" surrounded by percent signs (%) and ampersands (&). These &% and %& delimiters indicate a substitution variable. The calling program will need to substitute the user ID and password for the variables when it sends the System-Link request.

The third section is the heart of the System-Link request. This section defines the action (display, create, etc.) and the object, as well as the fields involved. If specific attributes are not selected, the fields will be defined using custom definitions. When the **System-Link Request** menu option is selected from a list window, the view, subset, and sort definitions used at the time will be included in the request. This means that the System-Link request will retrieve the same data that you see in the view or card file. User input for prompted subsets becomes part of the System-Link request.

In the illustration shown above, the view definition is *General* and the sort definition is *(default)*. When the **System-Link Request** option was selected from the menu, no subset was being used.

Every custom definition has an up-to 32-character token ("*objectId*") that is the internal identifier for that definition. Using object IDs instead of names can make requests language-independent. This is useful when using one System-Link request to support users with multiple languages on the same environment. For example, you can specify the correct definition whether it is called Default or *Valor por omission* or *Valeur par défaut*.

Alternately, using the name of the definition instead of the object ID can help create a System-Link request boilerplate for use across multiple environments. Each custom definition will have its own object ID. Therefore, two definitions residing in different environments can be given the same name. By separate handling of the components of a request document that are environment-specific (for example the Login action), one can minimize the number of System-Link request documents required in a multi-environment deployment.

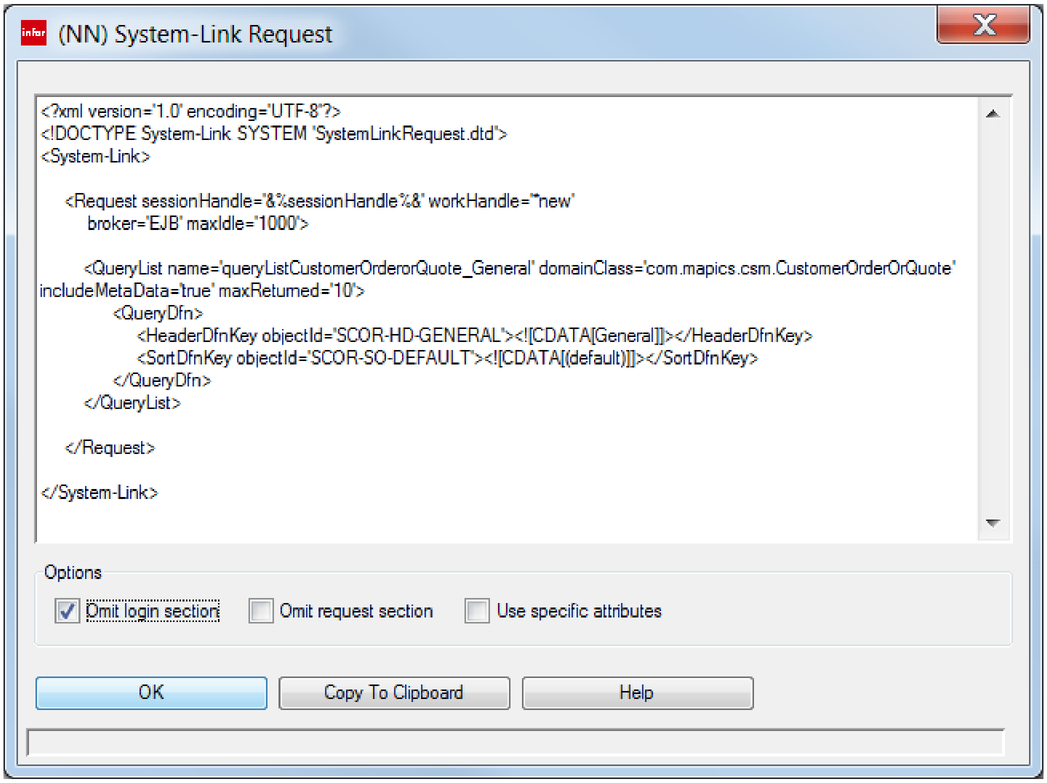
Note that the System-Link request generator lists both an object ID and a name for every definition included. If given both identifiers for a definition, System-Link will use the object ID.

The biggest advantage of using custom definitions to define System-Link requests is that the requests become very easy to change. For example, adding a field to a view will cause any System-Link request using that view to retrieve the new field. Technical support personnel will appreciate how much easier this makes the maintenance of System-Link requests.

There will be some cases where the System-Link request needs to be hard-coded. The PQL will specify the same fields as the view definition, the same filtering criteria as the subset definition, and the same sort order as the sort definition. There is no difference whatsoever in the data retrieved. As far as performance and other operational characteristics, there is no advantage or disadvantage to using PQL vs. custom definitions. Other than the maintenance issues mentioned, it is a matter of preference.

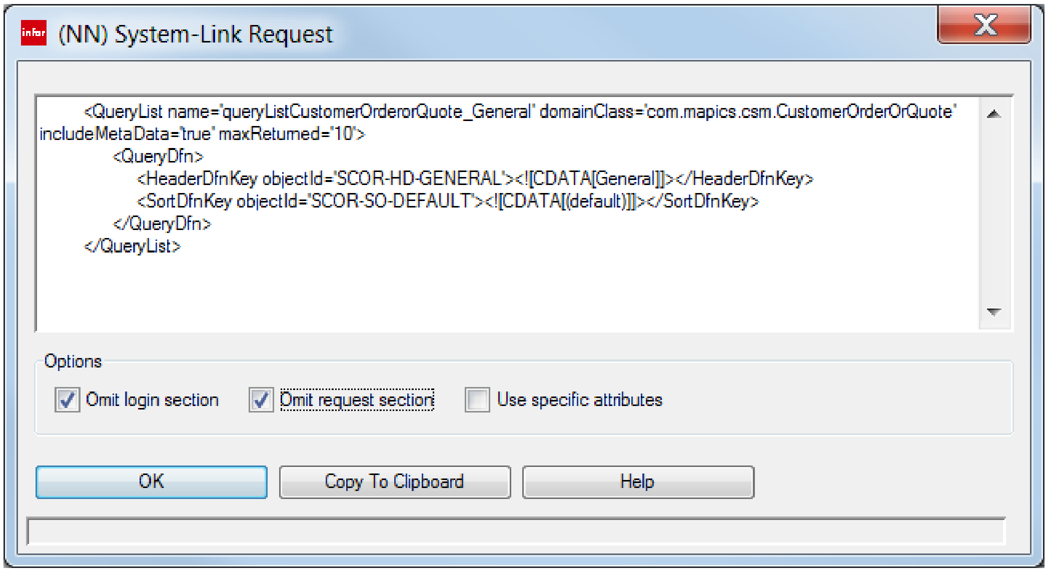
To use the System-Link request, click **Copy to Clipboard**. If you want to copy less than the entire XML request, you can highlight part of it and press Ctrl-C to copy the highlighted portion to the clipboard.

A successful login produces a session. A session can be left open to be reused for future requests, eliminating the need for the user to log in each time. To assist you in doing this, System-Link can strip off the login XML from the request. To do this, click the **Omit login section** checkbox. The System-Link Request window appears again without the login section:



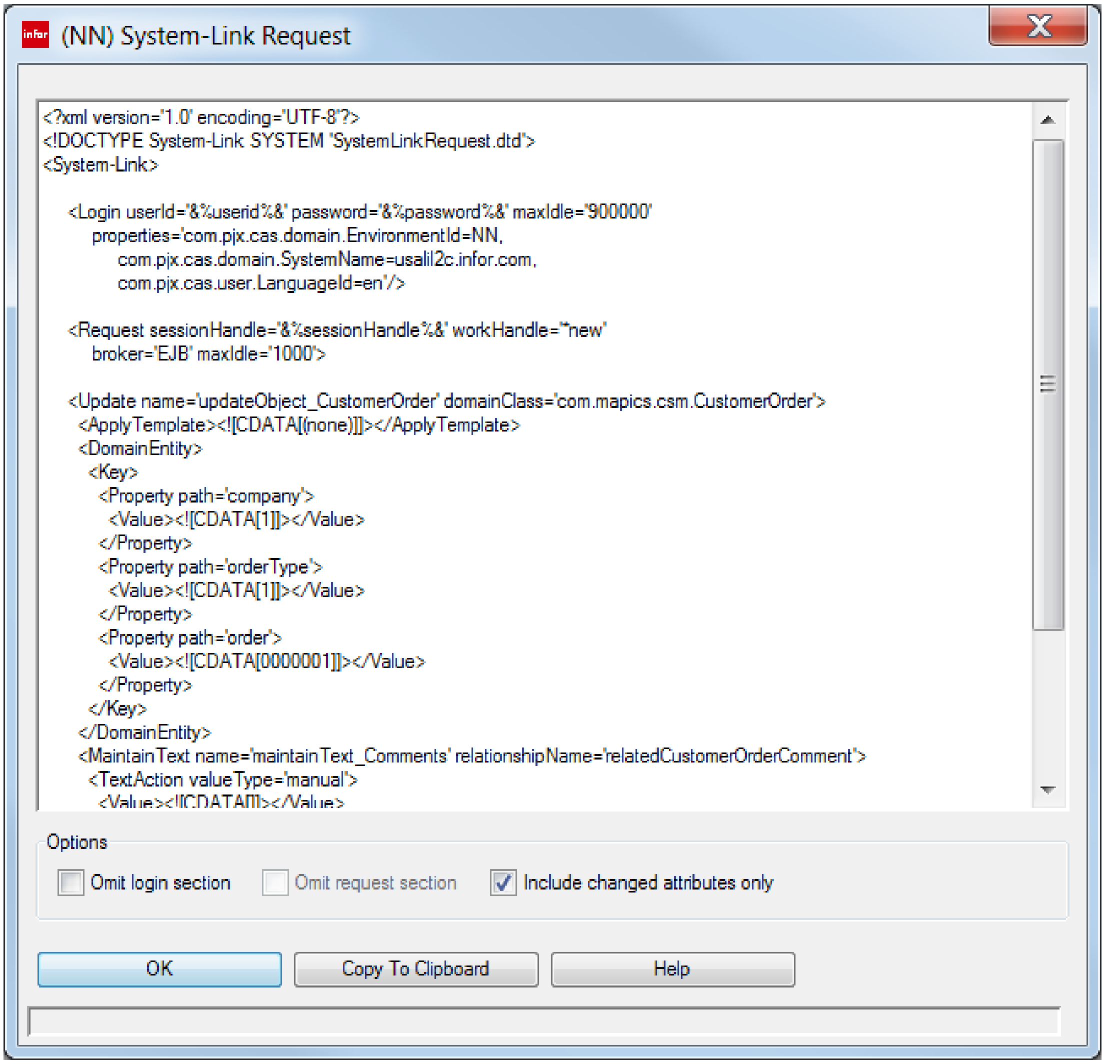
Some System-Link requests are simple, such as requesting a list of items or orders. Others can be complex, such as requesting a single order with all its line items, the releases for each one, the associated warehouse data and various overrides, holds, etc. Complex requests like this need to be assembled from parts of individually-generated requests.

When you omit the login section, the request generator asks if you want to strip off the request section as well. This leaves only the data section, which is all you need to assemble a complex request. To omit the request section of the XML, click **Omit request section** checkbox. The System-Link Request window appears again without the login and the request sections.



When selecting the **System-Link request** option from the menu on an object browser, the card file definition will be used to specify the list of fields to retrieve. Alternatively, PQL can specify the fields. In either case, all fields from all cards in the card file plus those in the banner area will be included.

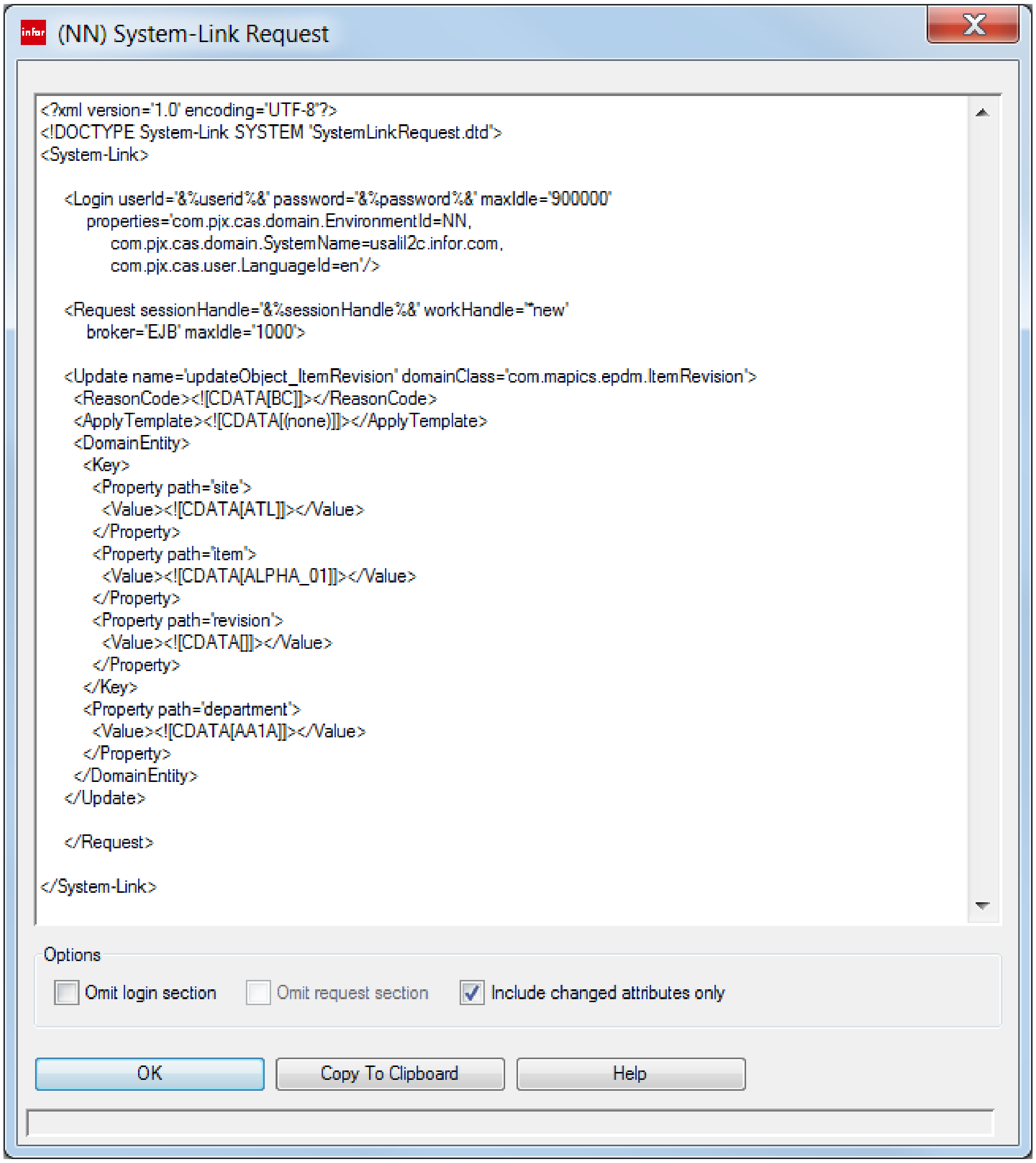
If the card file is in maintenance mode, the System-Link request will use a maintenance action such as create or update:



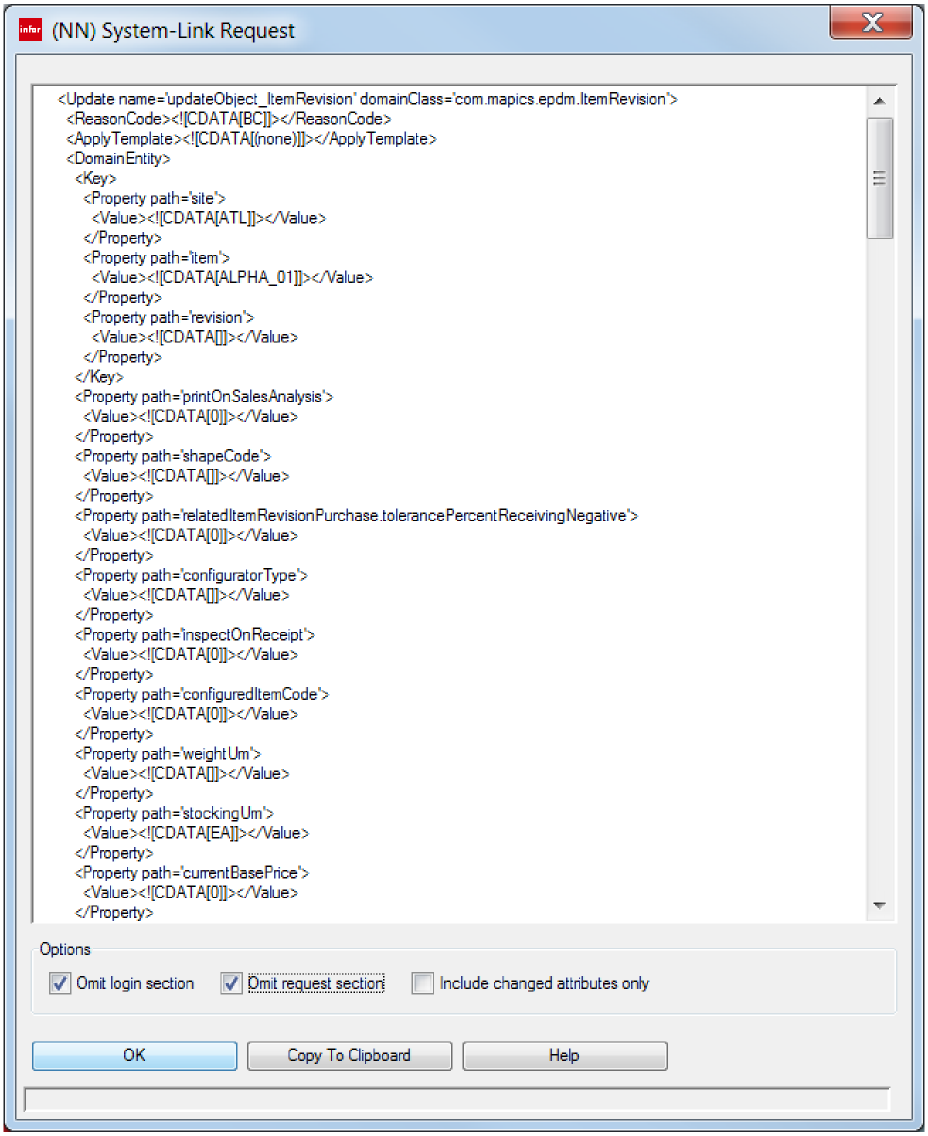
Note: The template value will be *(none)* even if the card file was in maintenance mode and a template was applied. The *<ApplyTemplate>* element is provided as a placeholder for you to be able to fill in the template name, if any.

Since the list of fields in a card file can be quite long, System-Link provides the ability to include just the changed fields in the request. The illustration below shows the update action for an item revision record where the department was changed to *AA1A*.

Note: The key fields, such as properties, are the only fields included other than *department*. (The department field was keyed into the card file but not entered, when the menu option was chosen to display this window. A reason code is also present, since reason code tracking was enabled.)



The illustration below shows what happens when **Include changed attributes only** check box is not checked. All fields on the banner and the cards in the card file are included. As you can see from the scroll bar, there is an enormous amount of information. If you need to see all of the attributes for some reason, clearing the **Include changed attributes only** check box will retrieve all attributes. By default, the **Include changed attributes only** check box is checked to only show changed fields.



The objective of the System-Link request generator is to provide you with a starting point for your System-Link request. You can copy it to the clipboard and paste it into your program, document, or the System-Link Simulator.

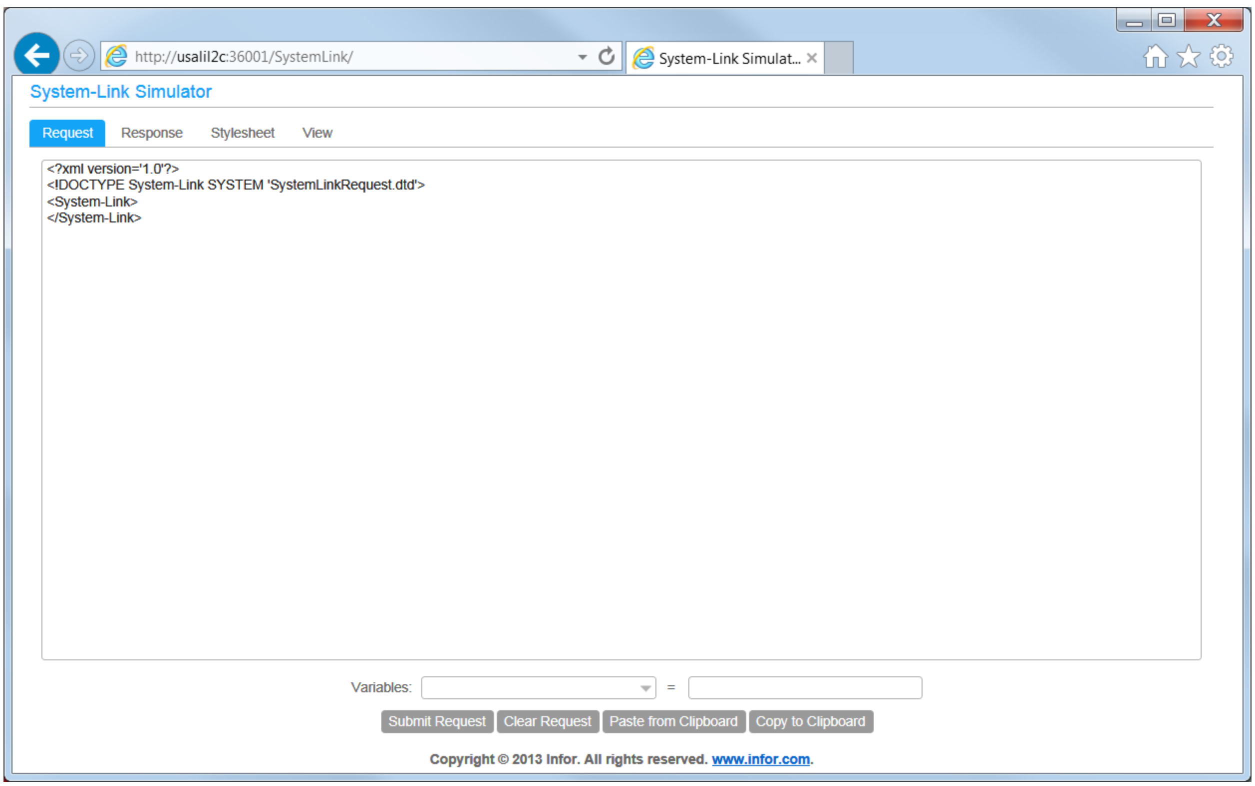
Chapter 8 Using the System-Link Simulator

8

The System-Link Simulator provides an easy way to test your System-Link request before putting it into production.

When you install System-Link, you can access the System-Link Simulator at *http://<your System i name>:<your secondary port>/SystemLink*. See Accessing System-Link via a Web Server.

On the Simulator, there are four tabs: **Request**, **Response**, **Stylesheet**, and **View**. Only one tab is displayed at a time. The tab currently displayed appears in a different color than the other tabs.



When the Simulator is first displayed, the **Request** tab is on top. To manage the work space, use the buttons located below the tabs:

Submit Request	Submits the System-Link request to the server. The Simulator automatically displays the **Response** tab while it waits for the System-Link response to return from the server.
Clear Request	Clears the work space on the **Request** tab. Without this button you would need to use the cursor to highlight the entire work space and press the Delete key.
Paste from Clipboard	Clears the workspace, then copies the request on the clipboard to the Simulator.
Copy to Clipboard	Copies the entire contents of the work space to the clipboard.

Note: When you want to copy less than the entire request, you can use your cursor to select (highlight) the part you want to copy. Place your cursor at the beginning of what you want to copy, then hold the Shift key down while using the keyboard cursor keys to highlight the text. Once the text is selected, you can use **Edit/Copy** or Ctl+C to copy it to the clipboard. Likewise, you can paste new portions into the window, rather than replacing the entire contents, by using Ctl+V.

When you want to copy the entire System-Link request, use **Copy to Clipboard** or highlight the entire request and use Ctl+C. Do not use any copy or selection options from your **Browser** menu. These will act upon the entire html document; including the graphics, the buttons, the header, etc.

Using variables in the Request

The System-Link Simulator supports the use of variables in the request. Variables allow you to insert data into a System-Link request before submitting it to the server. For example, you can have a System-Link request retrieve item data. The variable could be the item number that allows the same System-Link request to work for any item.

Variables are defined using this syntax:

`&%name%&`

where *name* indicates the variable name. Variable names can include any letter or number (no special characters, no blanks) and are case-sensitive. These are valid variable names:

userid, *variable1*, *DueDate*, *0001*.

They would appear in the XML as:

`&%userid%&`, **`&%variable1%&`**, **`&%DueDate%&`** and **`&%0001%&`**

Reserved variable names are *styleSheet, SystemLinkRequest,* and *SystemLinkRequestURL***.**

When you sign on to a web page using a user ID and password, the Simulator captures the user ID, password, and session handle as variables. The Simulator might capture additional variables, such as product or order numbers, dates, etc. When you use **send**, the variables are sent to the servlet (the URL includes *&userId=JSMITH*), which merges them with the request before submitting it to the server. In this way, the System-Link request is made flexible enough to support a wide variety of requests.

The **Variables** box to the right of this text area shows the variables and their values. When you first paste a System-Link request (one that includes the login section) into the Simulator, the *userid* and *password* variables appear in the **Variables** box.

To assign values to the variables, select the variable name in the **Variables** box. Two things happen when you do this:

1 A window appears below the variable name list where you can type in the value. As you type them in, the value appears next to the name in the **Variables** box.

2 In the text area, the variable is highlighted so you can find it in your XML. If you have the same variable in multiple places, only the first one is highlighted.

When you open the Simulator, there are no variable values. The values you enter are stored in the Simulator until you refresh it or close it. This means, for example, that you can use the same user ID and password for testing multiple System-Link requests without having to re-enter them each time.

When a System-Link request is created by Power-Link, the user ID and password are automatically assigned substitution variables (shown in bold in this example):

```
<Login userId='&%userid%&' password='&%password%&' maxIdle='900000'
    properties='com.pjx.cas.domain.EnvironmentId=6C,
        com.pjx.cas.user.LanguageId=enu'/>
```

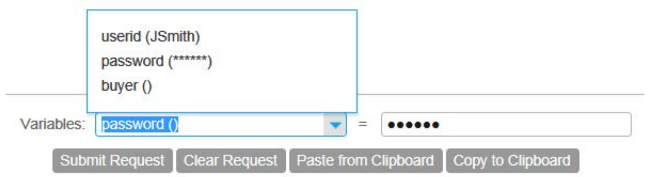
You can create additional substitution variables by entering the variable name surrounded by percent signs and ampersands. In this request, the *WHERE* clause asks for buyer 550 (shown in bold):

```
    <QueryObject name='queryObject_Buyer_Default'
 domainClass='com.mapics.pm.Buyer'    includeMetaData='true' >
            <Pql>
                <![CDATA[SELECT buyer,buyerName,buyerTelephone,fax,EMailAddress,
   buyerRating,department,buyerUserId, WHERE buyer='550']]>
            </Pql>
        </QueryObject>
```

The buyer number can be changed to be a variable called buyer (or anything else you choose). Variables are not shown in quotation marks. When you add the percent signs and ampersands, it looks like this (shown in bold) in the following example:

```
 <QueryObject name='queryObject_Buyer_Default' domainClass='com.mapics.pm.Buyer'
includeMetaData='true' >
          <Pql>
              <![CDATA[SELECT buyer,buyerName,buyerTelephone,fax,EMailAddress,
buyerRating,department,buyerUserId, WHERE buyer=&%buyer%&]]>
          </Pql>
     </QueryObject>
```

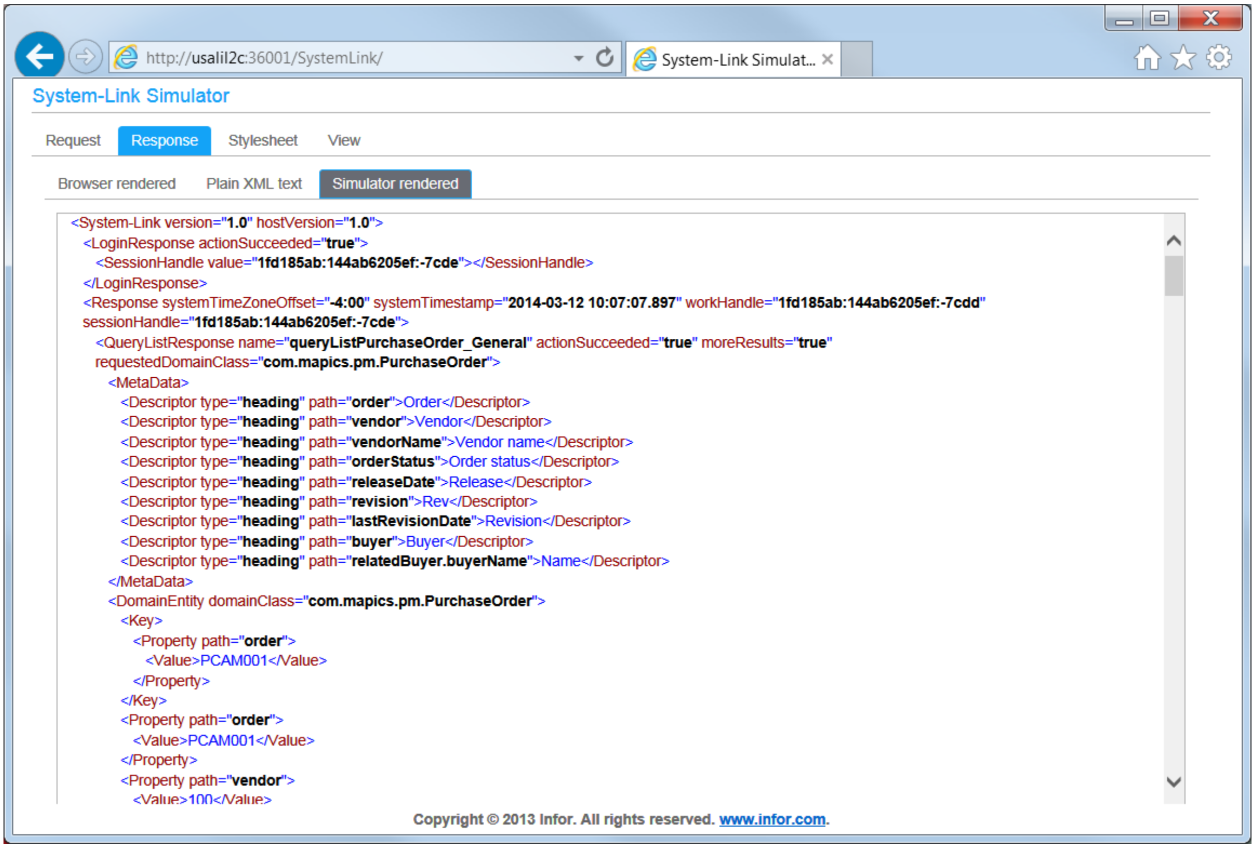
As soon as the last ampersand of ***&%buyer%&*** is entered, the new variable name appears in the **Variables** pull down. The new variable has exactly the same behavior as that described above for any other variable. The parentheses, next to the name in the **Variables** box, are empty because no value has been assigned. (The original 550 in the example above is ignored.) **Note:** The value for the password variable will always be obscured by the Simulator.



There is no limit to the number of variables you can have in your XML. The **Variables** list will expand and scroll as needed to show all present in the request document. There is no limit to how long a variable value can be in the Simulator. But, some browsers have a limit on how long a URL can be (for instance, 255 characters). Variables can be included in a URL. The variables plus the rest of the URL must fit within the limit determined by the browser.

The Response tab

The **Response** tab shows the System-Link response returned from the server. Below the **Response** tab are three additional tabs: **Browser rendered**, **Plain XML text**, and **Simulator rendered**.



Each additional tab displays the resulting response as:

- **Browser rendered** displays the response using the native mechanisms that the browser uses to display any XML file. The content under this tab is highly-dependent upon the browser (for example, Internet Explorer, Firefox, Chrome) and version.
- **Plain XML text** displays the response as unformatted XML text as returned by System-Link.
- **Simulator rendered** applies some pretty print formatting to the response, making the document easier to read.

The content under all additional tabs are blank when you first start the Simulator. The only way to get a System-Link response is to submit a System-Link request.

Notes:

1 You cannot edit the XML on any tabs under **Response** ; but, all or part of the XML can be highlighted using the cursor to copy it to the clipboard.

2 Depending upon browser and version, the XML preamble ("<?xml version="1.0" encoding="UTF-8"?> ") may or may not be displayed under one or more tabs under **Response**. The preamble is

a standard part of any XML document, and should be added if the tab contents are to be copied and used directly.

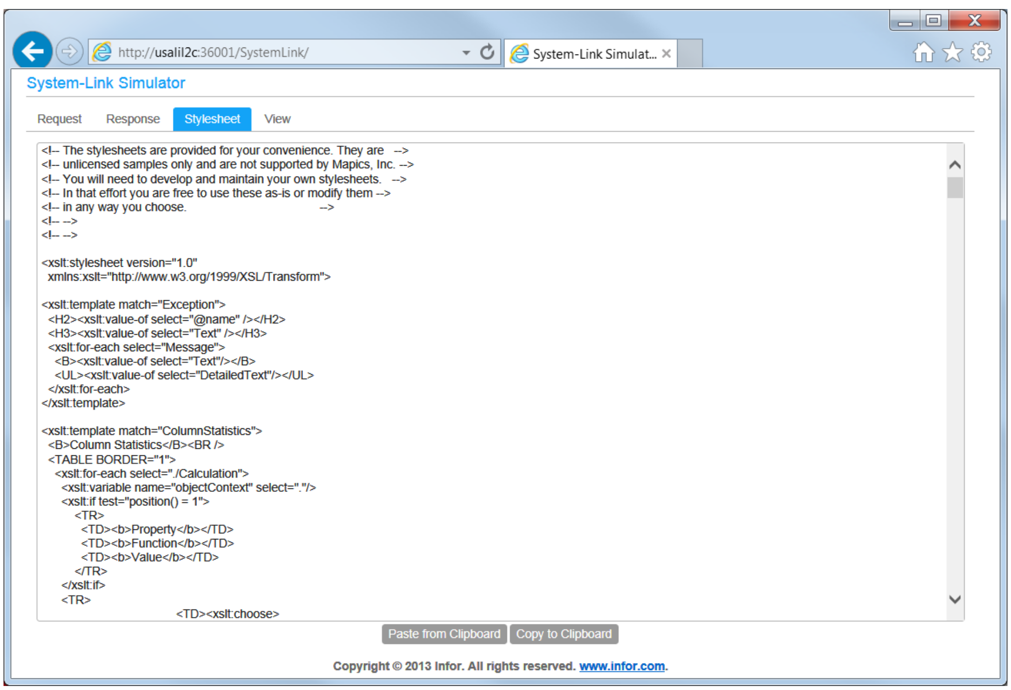
3 Depending upon browser and version, the content displayed under one or both rendered tabs may be dynamic, allowing sections of the response document to be expanded or collapsed as desired to aid viewing. Any additional text (for example, "+" and "-" symbols) are not part of the response document, and are not syntactically-correct XML. When in doubt, the **Plain XML text** tab contents will show the valid XML for the response (potentially subject to Note 2 above).

You can look at the response to verify that it is displaying what you want. This is the XML that is returned to your program. If the response is not what you want, you can switch back to the **Request** tab to change the System-Link request. You can then resubmit the request and check the response on the **Response** tab again. You can do this as many times as necessary.

The **Response** tabs will show any errors. Errors, due to mistakes in the request, can be corrected; and the request can be resubmitted to get a new response. You can also force errors intentionally, for example to test application edits.

The Stylesheet tab

The **Stylesheet** tab displays the XSL stylesheet. The **View** tab (described later) uses this stylesheet to format the System-Link response.



The stylesheet is valuable in two ways:

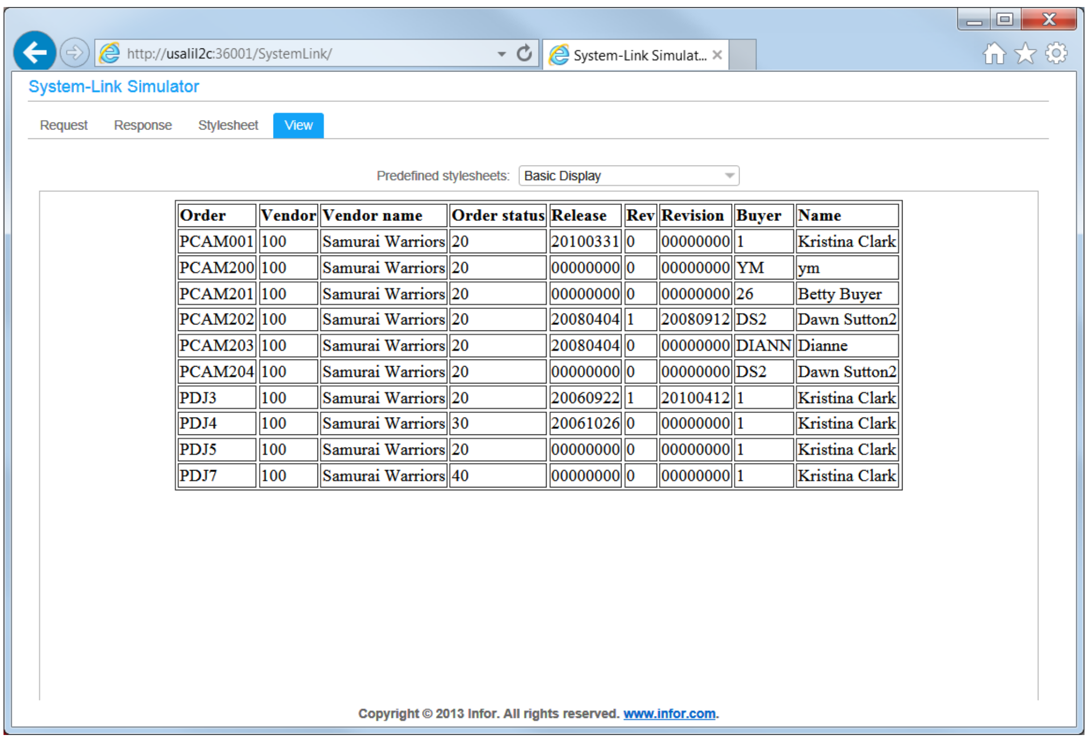
1 It formats the data in the System-Link response so that you can view it on the **View** tab. The stylesheet allows you to validate the XML request using the **View** tab, therefore, making it easier than examining the data in the System-Link response. The shipped stylesheets are used for this purpose.

2 It provides a way to test stylesheets used with web applications. The best characteristics of a stylesheet depend on the amount and type of data presented. The Simulator allows a programmer to paste in a stylesheet to see how it functions when used against an actual System-Link request. You can use the Simulator to make on-the-fly changes to the stylesheet and view the results. A satisfactory stylesheet can be copied from the Simulator to the clipboard and can be pasted into an XSL document.

Two basic stylesheets are shipped with the Simulator, as a convenience to the programmer. They are provided as-is and are not supported by Infor.

The View tab

The **View** tab shows how the XML from the **Response** tab looks after it is transformed by the XSL stylesheet defined under the **Stylesheet** tab.



Order	Vendor	Vendor name	Order status	Release	Rev	Revision	Buyer	Name
PCAM001	100	Samurai Warriors	20	20100331	0	00000000	1	Kristina Clark
PCAM200	100	Samurai Warriors	20	00000000	0	00000000	YM	ym
PCAM201	100	Samurai Warriors	20	00000000	0	00000000	26	Betty Buyer
PCAM202	100	Samurai Warriors	20	20080404	1	20080912	DS2	Dawn Sutton2
PCAM203	100	Samurai Warriors	20	20080404	0	00000000	DIANN	Dianne
PCAM204	100	Samurai Warriors	20	00000000	0	00000000	DS2	Dawn Sutton2
PDJ3	100	Samurai Warriors	20	20060922	1	20100412	1	Kristina Clark
PDJ4	100	Samurai Warriors	30	20061026	0	00000000	1	Kristina Clark
PDJ5	100	Samurai Warriors	20	00000000	0	00000000	1	Kristina Clark
PDJ7	100	Samurai Warriors	40	00000000	0	00000000	1	Kristina Clark

This is the easiest way to see whether your System-Link request or stylesheet needs to be changed. If the data is not what you expected, the System-Link request on the **Request** tab can be adjusted and resubmitted.

When the **View** tab displays problems with the stylesheet, you can change it on the **Stylesheet** tab. The query does not need to be resubmitted; changes to the stylesheet are immediately apparent in the **View** tab.

The stylesheets shipped with the Simulator displays a simple request containing a QueryObject action as a corresponding list of attributes. The stylesheets display a simple request containing a QueryList action as a corresponding table of attributes. Simple requests containing both a QueryObject and QueryRelationship actions display the object data followed by list data.

Note: The stylesheets shipped with the Simulator do not display deeper-nested relationship requests (that is tree structures). System-Link itself, whereas, will support an unlimited number of nested relationships under a given QueryObject request.

The System-Link Simulator is a convenient tool to test a System-Link request; although the Simulator is not required to process System-Link requests, nor is the Simulator intended to be used as a System-Link frontend in a production setting. Production implementations using System-Link should bypass the Simulator and contact the System-Link servlet directly.

The responsibility of the programmer is to construct a valid System-Link request appropriate to their requirements. The Simulator can assist in these ways:

- Web programmers can copy/paste a tested System-Link request from the Simulator to a web page. The web page needs to provide substitution variables, such as user ID, password, customer number, item number, etc. Web programmers can also use the Simulator to test the XSL (the stylesheet) rather than their System-Link requests.
- Server programmers can create programs that produce requests to be sent to System-Link. The Simulator can test such programs by processing the XML output. When the Simulator displays problems with a System-Link request, the program producing the XML needs to be modified accordingly. Thus, the Simulator provides programmers a way to test server-to-server programs.

Chapter 9 Examples

9

This chapter illustrates the System-Link syntax using typical business examples. Each example starts with the Power-Link list or object windows to show the business data in context. Once the view or card file has been described, it is followed by the System-Link request that would request the business data seen in the Power-Link window. A detailed discussion deconstructs the request in order to clarify each the role of the section in requesting the business data. Finally, the System-Link response is shown, with an equally detailed deconstruction of its various sections.

The examples are meant to be studied in order. Certain System-Link elements are used in some or all of the examples, but they are described only in the first example in which they appear. Subsequent examples consider only the new features. Note that only a subset of the entire System-Link functionality is presented here. As much as possible, System-Link maintains consistent syntax and behavior across action types. After becoming familiar with the general approach, one can find more examples and details in the *IDF System-Link Reference Guide*.

To assist you in finding various features, this table lists the first example where a feature is described:

Component	Example
Login element	Example 1: Query a list (Items)
Session handle	Example 1: Query a list (Items)
QueryList element	Example 1: Query a list (Items)
FetchNext element	Example 1: Query a list (Items)
PQL	Example 1: Query a list (Items)
Using view, subset, and sort definitions instead of PQL	Example 1: Query a list (Items)
MetaData element	Example 1: Query a list (Items)
DomainEntity element	Example 1: Query a list (Items)
Key property	Example 1: Query a list (Items)
Skipping login	Example 2: Query an object (Customer)
QueryObject element	Example 2: Query an object (Customer)
Key property (multi-part key)	Example 2: Query an object (Customer)

Component	Example
Properties Dates Decimals Boolean	Example 2: Query an object (Customer)
Using a card file instead of PQL	Example 2: Query an object (Customer)
QueryRelationship element	Example 3: Query multiple objects (Manufacturing Purchase Order)
Substitution variables	Example 3: Query multiple objects (Manufacturing Purchase Order)
Create element ReasonCode element ApplyTemplate element SourceObject element	Example 4: Create or copy a simple object (Warehouse)
ValueRef element	Example 5: Create multiple simple objects together (Warehouse with Address)
Update element	Example 8: Update (PO line)
Delete element	Example 9: Delete (Warehouse)

Spacing used in Examples

The XML shown in the examples may not always *exactly* match what you see in the Simulator using some browser versions. These points should be understood:

a) Generally, vertical space and horizontal space are ignored by the parser. For example, this request fragment indicates the value of the customer field is “101”:

```
<Property path='customer'>

    <Value>101</Value>

<Property>
```

The same XML could have the horizontal space eliminated…

```
<Property path='customer'>

<Value>101</Value>

<Property>
```

or, the vertical space eliminated:

```
<Property path='customer'><Value>101</Value><Property>
```

All three of these illustrations are identical to the parser. The examples in this chapter show XML with whatever spacing best illustrates the point being made.

b) This rule does not apply to non-element text values specified between the starting and ending tags (for example, text data associated with the Value element). In these cases, the parser will consider everything between the start and end tags to be intentional. In this fragment the value is “Example”:

```
<Value><![CDATA[Example]]></Value>
```

But, in this XML the value is “(new line)(space)(space)(space)Example(new line)”:

```
<Value>

   <![CDATA[Example]]>

</Value>
```

For this reason it is important to have only what you intend with plain text data between start and end tags, particularly in any place where a generated System-Link request includes the CDATA escape sequences above. System-Link respects this rule in the response, but some browser versions may add new lines to make the document easier to read. This is purely a display feature; the underlying response XML is unchanged.

c) In a similar fashion, some browser versions will sometimes add leading and trailing blanks to the significant characters in a CDATA section. The XML may be this:

```
<![CDATA[101]]>
```

But, these browser versions may display it like this:

```
<![CDATA[  101  ]]>
```

Again, this is a display-only feature. The underlying XML is unchanged.

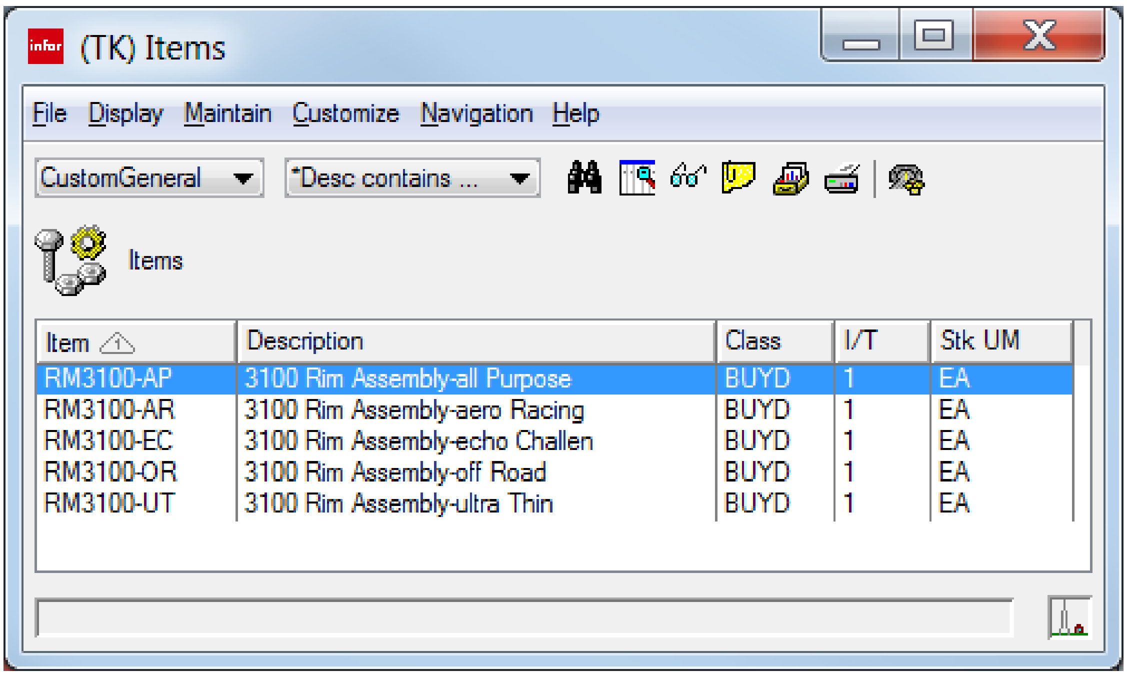
Example 1: Query a list (Items)

The screen shot shows a list of five items. This example includes the custom definitions as described below.

View Item, Description, Item class, Item type, and Stocking unit of measure

Subset All items containing “3100 Rim” in the description

Sort Item number in ascending sequence



XML Request

The System-Link request generated for the list window above looks like this:

```
<?xml version='1.0' encoding='UTF-8'?>

<!DOCTYPE System-Link SYSTEM 'SystemLinkRequest.dtd'>

<System-Link>

     <Login userId='&%userid%&' password='&%password%&' maxIdle='900000'
            properties='com.pjx.cas.domain.EnvironmentId=TK,
                        com.pjx.cas.domain.SystemName=usalil02.infor.com,
                        com.pjx.cas.user.LanguageId=en'/>

     <Request sessionHandle='&%sessionHandle%&' workHandle='*new'
              broker='EJB' maxIdle='1000'>
```

```
        <QueryList name='queryListItem_CustomGeneral'
                   domainClass='com.mapics.epdm.Item'
                   includeMetaData='true' maxReturned='10'>

              <Pql>

                   <![CDATA[SELECT item,description,itemClass,itemType,
                                   stockingUm WHERE (description LIKE
                                   '%3100 RIM%') ORDER BY item]]>

              </Pql>

        </QueryList>

    </Request>

</System-Link>
```

The request can be broken down into sections, shown below. The XML lines will be shown after the section heading, followed by a discussion of what the XML means.

Request Header

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE System-Link SYSTEM 'SystemLinkRequest.dtd'>
```

This entire System-Link request document will be sent to the XML parser. The parser needs to know what level of XML to expect. Therefore, the first line in this example tells the parser that the following XML will be compliant with XML 1.0 or higher. It also specifies the character set to use.

The second line identifies the DTD where the request syntax is specified. This DTD document was installed with System-Link.

System-Link element

```
<System-Link>
```

This specifies the beginning of the System-Link document to the parser. There is another element at the end to indicate that there is no more XML to follow for this System-Link document.

Login element

```
<Login userId='&%userid%&' password='&%password%&' maxIdle='900000'
       properties='com.pjx.cas.domain.EnvironmentId=TK,
                   com.pjx.cas.domain.SystemName=usalil02.infor.com,
                   com.pjx.cas.user.LanguageId=en'/>
```

The generated request adds substitution variables for the user ID and password values. If we want to run the request under user “JSMITH”, with password “1X2Y3Z”, the first line of the *Login* element would look like this:

```
<Login userId='JSMITH' password='1X2Y3Z' maxIdle='900000'
```

The maximum time the resulting session can be idle is 900,000 milliseconds, (equivalent to 900 seconds or 15 minutes). This request will be run against the “TK” environment on the “usalil02.infor.com” machine, and the language is “en” (English).

The user will experience a significant delay while the host system checks security and (if valid) creates a session. This delay can and should be avoided for future requests by skipping the login section and specifying the session handle in the subsequent Request element. The session handle is obtained from a previous System-Link response to a successful login.

Notice the forward slash (/) at the end, right before the greater-than sign (>). This indicates that there will not be a separate Login end tag.

Request element

```
<Request sessionHandle='&%sessionHandle%&' workHandle='*new'
         broker='EJB' maxIdle='1000'>
```

As with the user ID and password, the generated request also has a substitution variable in place for the session handle. For request documents containing a *Login* action, the session handle value must be “*current”. For those documents reusing a previously-created session, the session handle value will be that which was returned by the previous *LoginResponse*. Since our example request document has a *Login* action, the corresponding *Request* element would look like this:

```
<Request sessionHandle='*current' workHandle='*new'
         broker='EJB' maxIdle='1000'>
```

This *Request* section also uses a brand-new work area, as indicated by “*new” for the *workHandle* value. As mentioned in “The System-Link Protocol in Detail”, each *Request* section must have a work area to work in. Independent of its enclosing session, the work area can also expire. Thus, the System-Link request in our example specifies a maximum idle time for this work area of 900,000 milliseconds, or 15 minutes.

The value for the *broker* attribute is fixed at “EJB”. Other brokers may be available at a future date.

QueryList element

```
<QueryList name='queryListItem_CustomGeneral'
           domainClass='com.mapics.epdm.Item' includeMetaData='true'
           maxReturned='10'>
```

Since we want a list of objects, the System-Link request specifies a *QueryList* action. If we were asking for one object (that is a card file or details), the System-Link action would be *QueryObject*.

As with all System-Link actions, this *QueryList* action requires a name. Note that the name can be any sequence of characters valid in an XML attribute. System-Link uses the name to correlate the action with its results in the response document, as well as to retrieve the results of the action should they be used in subsequent actions (for example, *FetchNext*, *ValueRef*). In our example, the generator gave us a name of “queryListItem_CustomGeneral”. The name is a composite of the query type (list vs. object) plus the object being queried (Item) plus the view name (CustomGeneral). If you limit the number of records to be returned in one request, which is a good idea, the query name can also simplify a request for additional records (by using *FetchNext*).

Since the list is a list of items, the domain class is Item. In Java, the package must also be specified along with the name. (A package is similar to a path.) The package for Item is “com.mapics.epdm.” This means that request must ask for “com.mapics.epdm.Item” in order to query the Item object. Alternately, the “class mnemonic” (the legacy abbreviation) may also be used for that business class, by specifying *classMnemonic* instead of *domainClass* in the action. In this case, the class mnemonic would be “ITM”.

The attribute metadata, headings in the case of a *QueryList* action, will be returned in the System-Link response (*includeMetaData=’true’*).

The server will return only ten records at the most (*maxReturned=’10’*). In our example response (shown below) we only received five because that is all that were present. If there had been more than ten we would have received the first ten. This parameter is useful to cap the amount of data being returned to the requesting program. If the requestor is a user, (as opposed to a server program) it is usually recommended that you send only the amount of data that a user can reasonably comprehend at one time. A display might show fifty records per page, and thus only the first fifty records would be requested. You can write the System-Link request so that the user can keep asking for more, with each subsequent System-Link request limited by this parameter.

PQL element

```
<Pql>
     <![CDATA[SELECT item,description,itemClass,itemType,
                     stockingUm WHERE (description LIKE
                     '%3100 RIM%') ORDER BY item]]>
</Pql>
```

This section specifies the details of the data request. It maps closely to the view, subset, and sort of the original Power-Link list window. As mentioned in the “The System-Link Protocol in Detail”, the syntax of this statement should be somewhat-familiar to anyone with knowledge of SQL.

Select This is similar to a view, since it specifies the attributes that will be provided for each record. The attributes are not exactly like the attribute descriptions in Power-Link, because they must follow different naming rules. For example, they must not have blanks, they must be unique, and they have specific rules about upper/lower case. Nevertheless, they are still recognizable as the columns from the original view.

Where This is similar to a subset. The subset we used in the original list window was all items with a description that contained PC*.* Here you can see that PC is preceded and followed by a percent sign (%) indicating a wildcard (like an asterisk in Windows).

Order by This is similar to a sort. The default order is ascending, so it is not specified. Had it been descending, it would have DESC at the end, as in ORDER BY item DESC.

Exclude System-Link honors IDF security, including content security. To improve performance the records can be excluded on the host by adding EXCLUDE UNATHORIZED to the end of the PQL. Without the *EXCLUDE* clause a Not authorized message will be returned for each record to which the requestor is not authorized.

QueryDfn element

The following is an example of a *QueryDfn* section that could be used in place of the *Pql* to yield the same result. (Note that we chose to specify the definition names in this example, rather than the object IDs. This choice is purely for illustration, but either form could be used.)

```
<QueryDfn>

  <HeaderDfnKey
clientClass='com.mapics.epdm.Item'><![CDATA[CustomGeneral]]></HeaderDfnKey>

  <SubsetDfnKey clientClass='com.mapics.epdm.Item'><![CDATA[Desc contains
...]]></SubsetDfnKey>

  <PromptedSubsetValues>

    <CriteriaSpec path='description'>

      <Value><![CDATA[3100 Rim]]></Value>

    </CriteriaSpec>

  </PromptedSubsetValues>

  <SortDfnKey clientClass='com.mapics.epdm.Item'><![CDATA[(default)]]></SortDfnKey>

</QueryDfn>
```

The *QueryDfn* element is an alternative to the *PQL* element. It yields the same results using custom definitions.

HeaderDfnKey This replaces the *SELECT* clause by identifying the view to use.

SubsetDfnKey This replaces the *WHERE* clause by identifying the subset to use. A prompted subset will require an additional *PromptedSubsetValues* section to supply the prompt value(s).

SortDfnKey This replaces the *ORDER BY* clause by identifying the sort to use.

More details on how to create this element can be found in the “Building a System-Link request using Power-Link”.

System-Link Response

The System-Link response is what will be returned to the calling program. When using the Simulator, depending upon the capabilities of the browser, one has the ability to expand or collapse nodes within the response XML. The minus sign (-) appears next to a node that can be collapsed, while the plus sign (+) appears next to a node that can be expanded. The XML on the Response tab is first displayed with all nodes expanded. In order to make the XML in which you are interested easier to comprehend, it is often helpful to collapse the other nodes.

The System-Link response for the previous request is shown below as returned by Internet Explorer, with some whitespace added for clarity. Nodes with a plus sign have been collapsed. (Minus signs on expanded nodes have been removed from the example.)

```
<?xml version="1.0" encoding="UTF-8" ?>

<System-Link version="1.0" hostVersion="1.0">

  <LoginResponse actionSucceeded="true">

    <SessionHandle value="-63121ea5:140a6658bdf:-7ec8" />

  </LoginResponse>

  <Response sessionHandle="-63121ea5:140a6658bdf:-7ec8"
            workHandle="-63121ea5:140a6658bdf:-7ec7"
            systemTimestamp="2013-08-22 15:17:01.432"
            systemTimeZoneOffset="-4:00" hasErrors="false"
            hasWarnings="false">

    <QueryListResponse name="queryListItem_CustomGeneral"
                        requestedDomainClass="com.mapics.epdm.Item"
                        actionSucceeded="true" moreResults="false">

     +<MetaData>

     +<DomainEntity domainClass="com.mapics.epdm.Item">

     +<DomainEntity domainClass="com.mapics.epdm.Item">

     +<DomainEntity domainClass="com.mapics.epdm.Item">

     +<DomainEntity domainClass="com.mapics.epdm.Item">

     +<DomainEntity domainClass="com.mapics.epdm.Item">

    </QueryListResponse>

  </Response>

</System-Link>
```

Response Header

```
<?xml version="1.0" ?>

<System-Link version="1.0" hostVersion="1.0">
```

The System-Link response header performs the same functions as the XML request header. This is because the response may be passed to other systems that do not have access to the request. Thus, the first line says that the following XML will be compliant with XML 1.0 or higher.

Login response element

```
<LoginResponse actionSucceeded="true">
  <SessionHandle value="-63121ea5:140a6658bdf:-7ec8" />
</LoginResponse>
```

The *LoginResponse* shows that the user ID and password were valid. The result is a session handle that will be used for this request. Had the user ID or password not been valid the response would have been limited to an error message.

The session will remain idle for 15 minutes, since that is the maximum idle time we specified at login. Note that the idle time is not cumulative: using a session resets its idle timer. We can avoid the login delay by using this session handle for additional requests. We would simply omit the login action and include the session handle value in place of "*current" in our request element.

Response element

```
<Response sessionHandle="-63121ea5:140a6658bdf:-7ec8"
          workHandle="-63121ea5:140a6658bdf:-7ec7"
          systemTimestamp="2013-08-22 15:17:01.432"
          systemTimeZoneOffset="-4:00" hasErrors="false"
          hasWarnings="false">
```

The session handle in the *Response* element is the same as that returned in the login response. The work area is like a session within a session. Multiple work areas are useful for maintenance, especially for complex objects where you want to send a header record and multiple detail records all together. Generally, for inquiry the entire response will be for one session and one work area.

The *systemTimestamp* and *systemTimeZoneOffset* attributes reflect the current time on the machine running the SLS instance, as well as its offset from GMT.

The *hasErrors* and *hasWarnings* attributes indicate whether any warnings or errors were encountered for any actions within that Request section.

QueryListResponse element

```
<QueryListResponse name="queryListItem_CustomGeneral"
                   requestedDomainClass="com.mapics.epdm.Item"
                   actionSucceeded="true" moreResults="false">
```

What follows is the response to the "queryList_Item_Sample" action for the Item object. *moreResults = "false"* means that we have all the records that met the selection criteria. The request specified no more than ten records in the response. Ours has five. If *moreResults* were "true", we could use the *FetchNext* element, described later, to display any or all of the remaining results. (Note that the original *QueryList* action did not specify a value for the *retainResults* attribute. However, for *QueryList* the default value for this attribute is "true". Any remaining results will therefore by default be available for a subsequent *FetchNext* action.)

A System-Link *Request* section can contain more than one *QueryList* action. Each will have a unique name. The System-Link response will provide the same name in each *QueryListResponse* so that the response can be associated with its *QueryList* request.

MetaData element

```
<MetaData>
```

The *MetaData* element indicates that the XML to follow is metadata: data that describes the actual business object data values. You only get metadata in the response if you ask for it in the *QueryList* element. Generally, you ask for metadata when you expect to process the response for direct display to the user (for example in a web page). If the results were instead being sent to another system (that is server-to-server), skipping the metadata would eliminate the overhead both to retrieve the metadata and to send it in the document.

Our original view had five columns. Each one is an attribute, so our System-Link request included five attributes. XML attribute names are similar to the Power-Link list attribute names, except that there are no spaces and the capitalization is different:

Attribute Description	Metadata Heading	XML attribute name
Item	Item	item
Description	Description	description
Item class	Class	itemClass
Item type	I/T	itemType
Stocking UM	Stk UM	stockingUm

Each attribute has its own *Descriptor* element in the metadata section. The following lines describe the metadata for the Item attribute:

```
<Descriptor type="heading" path="item"><![CDATA[Item]]></Descriptor>
```

This fragment says that the metadata text is a “heading”. (Another type could be “label”, which we will see in our next *QueryObject* example.) The *path* indicates which attribute this heading is for – in this case, the item attribute. Finally, the character data inside the *CDATA* section of the metadata for this attribute is Item, which corresponds to the column heading shown for that property in the Power-Link list browser. Therefore, Item is the heading for the item attribute. This may look redundant since the attribute and the heading are almost identical. Sometimes the difference between the heading and the attribute name is more pronounced, as in the case of Item type:

```
<Descriptor type="heading" path="itemType"><![CDATA[I/T]]></Descriptor>
```

In this case, “I/T” is the heading for the itemType attribute.

In the previous view of the response XML the metadata node was collapsed. It is shown expanded below. (Note that the browser added whitespace between the *Descriptor* start and end tags, as well as before and after each descriptor value string. This is a display artifact introduced by the browser, and is not present in the actual System-Link Response document.) You will see all five of the attributes listed. At the end of the *MetaData* section is the end tag:

```
</MetaData>
```

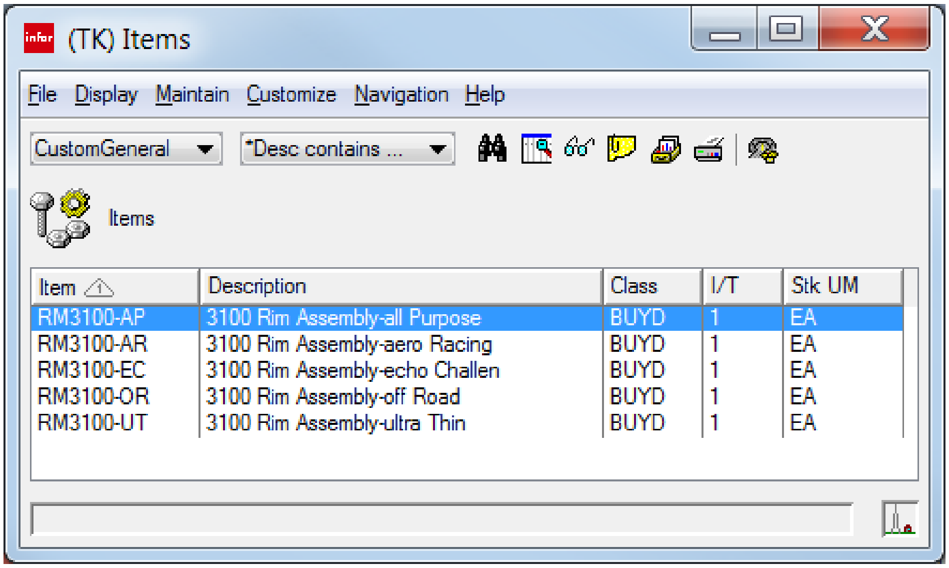
```
********   NOTE: The start of the XML document is not shown. ********

<QueryListResponse name="queryListItem_CustomGeneral"
                   requestedDomainClass="com.mapics.epdm.Item"
                   actionSucceeded="true" moreResults="false">

  <MetaData>

    <Descriptor type="heading" path="item">

      <![CDATA[ Item ]]>

    </Descriptor>
```

```
    <Descriptor type="heading" path="description">
      <![CDATA[ Description ]]>
    </Descriptor>
    <Descriptor type="heading" path="itemClass">
      <![CDATA[ Class ]]>
    </Descriptor>
    <Descriptor type="heading" path="itemType">
      <![CDATA[ I/T ]]>
    </Descriptor>
    <Descriptor type="heading" path="stockingUm">
      <![CDATA[ Stk UM ]]>
    </Descriptor>
  </MetaData>
  <DomainEntity domainClass="com.mapics.epdm.Item">
********   NOTE: The rest of the XML is not shown. ********
```

DomainEntity element

```
<DomainEntity domainClass="com.mapics.epdm.Item">
```

A *domain entity* is a business object, like Item, Customer, Order, or Vendor. This is the business data that your System-Link request is returning. Our System-Link request asked for item data, and this element signals that the request that follows will be from the Item object. Note our previous list window showing five items.



This is the data that our System-Link response will contain. Each line of data will have a *DomainEntity* element in the response. Since we are getting five records, we see five domain entity elements. Each one will be identical to the one shown above, but the keys and properties following them will vary based on the data in the records.

The System-Link response shown earlier had the *DomainEntity* elements collapsed. This fragment shows the XML for the first *DomainEntity* element:

```
<DomainEntity domainClass="com.mapics.epdm.Item">
  <Key>
    <Property path="item">
      <Value><![CDATA[RM3100-AP]]></Value>
    </Property>
  </Key>
  <Property path="item">
    <Value><![CDATA[RM3100-AP]]></Value>
  </Property>
  <Property path="description">
    <Value><![CDATA[3100 RIM ASSEMBLY-ALL PURPOSE]]></Value>
  </Property>
  <Property path="itemClass">
    <Value><![CDATA[BUYD]]></Value>
  </Property>
  <Property path="itemType">
```

```
    <Value><![CDATA[1]]></Value>
  </Property>
  <Property path="stockingUm">
    <Value><![CDATA[EA]]></Value>
  </Property>
</DomainEntity>
```

Each domain entity element starts with the key of the record. The fragment for this is shown below:

```
<Key>
  <Property path="item">
    <Value><![CDATA[RM3100-AP]]></Value>
  </Property>
</Key>
```

The sequence of statements provides a good illustration of the symmetrical structure of XML. Each element has a beginning and an end. In this case, you see the *Key* start tag to begin the list of key properties and the *Key* end tag at the bottom to signal the end of the key properties.

Between the key elements, you see a *Property* element. The property *path* is the attribute (notice the similarity to path in the metadata). In our example, this is item because that is the key of the Item object. For a multi-part key, we would have seen multiple *Property* elements, each showing a different part of the key. For example, if the object had been Item Revision we would have seen three *Property* elements: one for Site, a second for Item and a third for Revision.

The value of a property is shown between the *Value* tags. Since the key of the first record is "RM3100-AP" that is the value.

The key element defines the record that is to follow. The first record is as follows (the headings are shown for convenience):

Item	Description	Class	I/T	Stk UM
RM3100-AP	3100 RIM ASSEMBLY-ALL PURPOSE	BUYD	1	EA

Each attribute will be described using properties. For example, the property for the first attribute is shown below:

```
<Property path="item">
-    <Value><![CDATA[RM3100-AP]]></Value>
</Property>
```

Note that this is the same as the key field. However, there is no *Key* element here. The *Key* element defines the record. The *Property* element identifies the data. System-Link will not assume that key data is to be displayed. Often it is useful to not display the key…or not all of it. For example, many objects have company as part of the key. Organizations with only one company usually choose not to display it.

All four of the remaining *Property* elements (not shown) are listed: one after the other. Following the last one is the closing *DomainEntity* tag, signaling the end of the record.

```
</DomainEntity>
```

The next record then starts with a new *DomainEntity* tag. This pattern is repeated for the next record, and the next, until the last one. At the end of the System-Link response, you see the final closing *DomainEntity* tag…

```
</DomainEntity>
```

…followed by the closing *QueryListResponse* tag…

```
</QueryListResponse>
```

…followed by the closing *Response* tag…

```
</Response>
```

…followed by the closing *System-Link* element at the very end:

```
</System-Link>
```

FetchNext Request

A *queryList* request will return a list of records up to the number specified in the *maxReturned* attribute. In our example, if there had been fifteen records in our list, instead of five, the response would have returned only ten – the *maxReturned* value in our request. The appearance of *moreResults="false"*, which means there were no more records, would have instead be *moreResults="true"*, meaning that more records were available.

Usually we need to provide the user with the ability to get additional records. Allowing the user to repeat the same *QueryList* request would simply return the same data a second time. Instead, we need the ability to repeat the *QueryList* action (same object, same attributes, etc.) with the stipulation that it returns the next set of records. This is the purpose of the *FetchNext* element.

Below is a request fragment for the first ten items in the item master. The PQL contains no *WHERE* clause, meaning there is no subset. This is equivalent to the all records designation in the Power-Link list browser.

```
<QueryList name='queryList_Item_General'
           domainClass='com.mapics.epdm.Item' includeMetaData='true'
           maxReturned='10'>

  <Pql><![CDATA[SELECT item,description,itemClass,itemType,drawingNumber,
                stockingUm,department ORDER BY item]]>

  </Pql>
```

The example response fragment below shows the session handle, the work handle and the *moreResults* indicator with value of *"true".*

```
<Response sessionHandle="-63b0faa5:e53fe694be:-7f6e"
          workHandle="-63b0faa5:e53fe694be:-7f6c">

  <QueryListResponse name="queryList_Item_General"
                     requestedDomainClass="com.mapics.epdm.Item"
                     actionSucceeded="true" moreResults="true">
```

In order to get additional records, we must create a new request using the *FetchNext* element. The request must specify the same session handle and work handle (both shown in bold) as listed in the response to the earlier request.

```
<?xml version='1.0'?>

<!DOCTYPE System-Link SYSTEM 'SystemLinkRequest.dtd'>

<System-Link>

  <Request sessionHandle='-63b0faa5:e53fe694be:-7f6e'
           workHandle='-63b0faa5:e53fe694be:-7f6c' broker='EJB'
           maxIdle='7000000'>

    <FetchNext name='second_group' queryName='queryList_Item_General'

               maxReturned='20'/>

  </Request>

</System-Link>
```

The *FetchNext* element above has three attributes: a *name*, a *queryName* and *maxReturned*.

Attribute Name	Attribute Value	Description
name	second_group	The name allows the parser to pair the response with the FetchNext request.
queryName	queryList_Item_General	This tells the server to return the results of this previous query, starting with the next record.
maxReturned	20	The server will not assume that the maximum is the same as the earlier request. In this case, we are assuming that if the user is not satisfied with ten records we will be a little more aggressive with the next request and return twenty.

The *FetchNext* response is very close to the *QueryList* response.

```
<Response sessionHandle="-63b0faa5:e53fe694be:-7f6e"
          workHandle="-63b0faa5:e53fe694be:-7f6c">

  <FetchNextResponse name="second_group" queryName="queryList_Item_General"
                     actionSucceeded="true" moreResults="true">
```

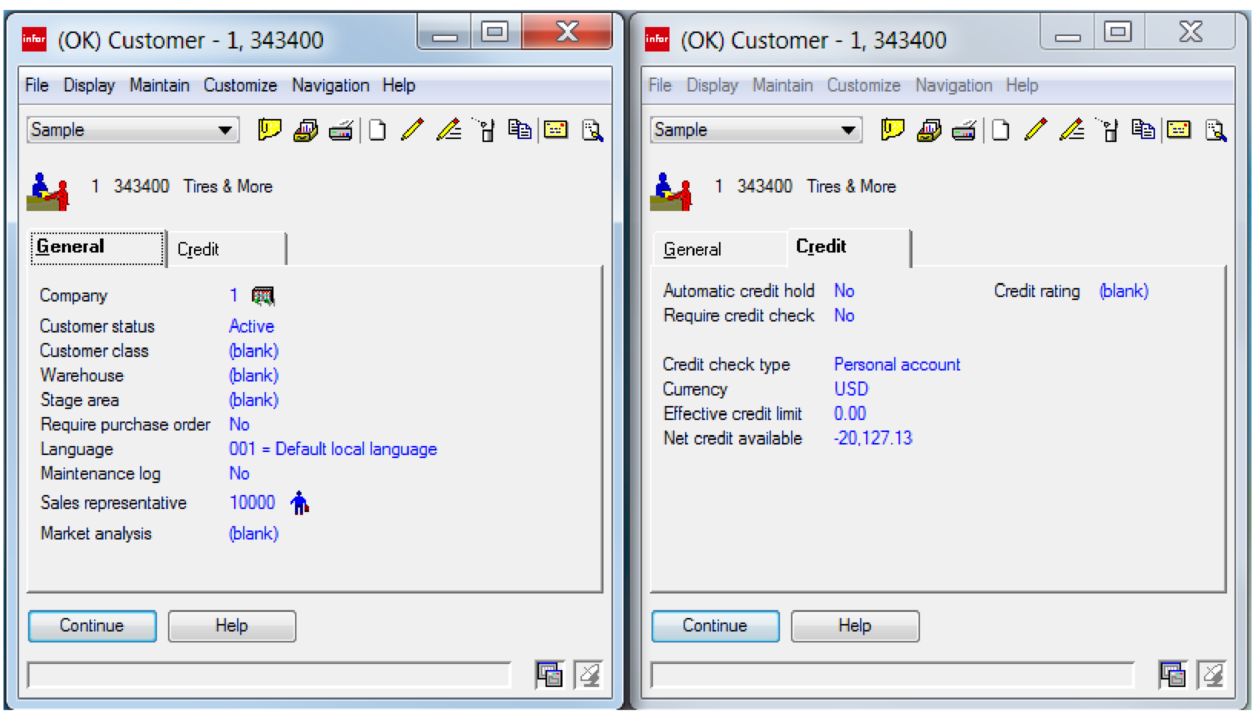
After the *FetchNextResponse* element, the *DomainEntity* elements begin. The *DomainEntity* structure is identical to that in the original *QueryListResponse*. We will get twenty, because there were at least twenty Items available, and because that was the maximum we asked for. The *moreResults="true"* tells us that we got the maximum…and there are still more for the asking.

Caution: the reason why we could use the *FetchNext* against the *QueryList* action as written is that a *QueryList* action has its *retainResults* attribute default to "true". If one knows that the results of a specific *QueryList* action will not be involved in a subsequent *FetchNext* request, then the *QueryList* can have its *retainResults* attribute explicitly set to "false". This will save some overhead if more entries are present than the value of *maxReturned*. But, in most cases a *QueryList* will involve one or more subsequent *FetchNext* actions to retrieve all of the data in a controlled fashion.

Example 2: Query an object (Customer)

System-Link Request

The previous example discussed a number of features that also apply to this example. These features include the login element, session handles, *PQL* or *QueryDfn* elements, and meta data. These topics will not be discussed in detail again. Refer to the previous example for illustrations of these features.



This example will request the record for one customer. Below we see details for one customer. The card file contains only two cards in order to simplify the example. Both cards are shown below:

The object has a two-part key:

Key field	Value
Company number	1
Customer number	343400

The first card has ten attributes. The second card has seven more, for a total of seventeen.

The *Request* section of the generated System-Link request document is shown below:

```
<Request sessionHandle='&%sessionHandle%&' workHandle='*new'
         broker='EJB' maxIdle='1000'>


    <QueryObject name='queryObject_Customer_Sample'
                 domainClass='com.mapics.csm.Customer'
                 includeMetaData='true'>

      <Pql>

           <![CDATA[SELECT company,customerStatus,customerClass,
                    warehouse,stageArea,requirePurchaseOrder,
                    language,maintenanceLog,salesrep,

                    marketAnalysis,
                    automaticCreditHold,requireCreditCheck,
      relatedPersonalAccount.relatedPersonalAccountSummary.creditCheckType,
                    netCreditAvailableCurrency,effectiveCreditLimit,
                    netCreditAvailable,creditRating WHERE
                    company = 1 AND customer = 343400]]>

      </Pql>

    </QueryObject>


</Request>
```

Note that the substitution variable for the session handle is provided. If we were using an existing session for this request (which is always suggested whenever possible), then the *Request* section would look something like this:

```
<Request sessionHandle='7f1ba3:e5347e3f6f:-7e13' workHandle='*new'
         broker='EJB' maxIdle='1000'>
```

This makes the request process more quickly as well as uses system resources more efficiently. Other than the session handle, the rest of the request is similar to that already described in the previous example.

QueryObject element

```
<QueryObject name='queryObject_Customer_Sample'
             domainClass='com.mapics.csm.Customer'
             includeMetaData='true'>
```

This *QueryObject* element is virtually identical to the *QueryList* element. The only visible difference is that it does not have a maximum return value, because it will return only one record. Unlike *QueryList*, the default *retainResult* value for *QueryObject* is false. This default would affect any subsequent value references that want to use the results of the *QueryObject* action. Since our example will not involve a value reference, this is the desired behavior.

PQL element

```
<Pql>

     <![CDATA[SELECT company,customerStatus,customerClass,
                     warehouse,stageArea,requirePurchaseOrder,
                     language,maintenanceLog,salesrep,marketAnalysis,
                     automaticCreditHold,requireCreditCheck,
relatedPersonalAccount.relatedPersonalAccountSummary.creditCheckType,
                     netCreditAvailableCurrency,effectiveCreditLimit,
                     netCreditAvailable,creditRating WHERE
                     company = 1 AND customer = 343400]]>

</Pql>
```

Again, the differences between this and the *QueryList* PQL are minor. The *SELECT* clause lists seventeen attributes: the ten from the first card and the seven from the second. (Note that the customer name attribute appears in the banner but not on the cards themselves. This attribute will therefore not be present in the generated PQL or the result of the *QueryObject* action.)

The *WHERE* clause contains the values for the key field. This will insure that only one record is retrieved.

There is no *ORDER BY* clause, since sort does not apply.

The remainder of the XML is composed of end tags.

Using custom definitions instead of PQL

In the PQL above, the *SELECT* clause lists each of the fields to be retrieved. The *WHERE* clause identifies the key field values of the record to be retrieved.

You can specify the key field values using the *Key* element, as shown below. The *Key* element would be used in place of the *PQL* element.

```
<Key>

  <Property path='company'>

    <Value>1</Value>

  </Property>

  <Property path='customer'>
```

```
        <Value>343400</Value>
      </Property>
    </Key>
```

Now that System-Link knows which record you want, you need to specify the data to be retrieved. This can be done by referencing a card file, using the *CardFileDfnKey* element immediately after the *Key* element. The card file is called Sample in the following example. (Note that as before we are referring to the card file by name rather than by object ID; either type of reference will work.)

```
<CardFileDfnKey
clientClass='com.mapics.csm.Customer'><![CDATA[Sample]]></CardFileDfnKey>
```

Everything from the <Key> start tag to the </CardFileDfnKey> end tag would replace the *PQL* element, that is everything from the <PQL> start tag to the </PQL> end tag inclusive. The rest of the XML is unchanged.

The advantage to this approach is that the System-Link results can be controlled very simply by customizing a card file. Thus, if an attribute is added to a card in the card file it will automatically be included in the next System-Link retrieval referencing that card file. The same goes for attributes removed from a card. Situations where this flexibility is too dynamic can still use PQL to create hardcoded requests.

System-Link Response

The example below will concentrate on those areas where the response for the *QueryObject* request differs from that for the *QueryList* request. Thus, the header and *Response* elements are skipped. All but one of the *MetaData* nodes and many of the *Property* nodes have been collapsed or omitted for the sake of brevity. The Key element under the *DomainEntity* node shows how multiple key properties are handled.

```
<QueryObjectResponse name="queryObject_Customer_Sample"
                     requestedDomainClass="com.mapics.csm.Customer"
                     actionSucceeded="true">
  <MetaData>
  + <Descriptor type="label" path="company">
  + <Descriptor type="label" path="customerStatus">
  + <Descriptor type="label" path="customerClass">
  + <Descriptor type="label" path="warehouse">
  + <Descriptor type="label" path="stageArea">
  + <Descriptor type="label" path="requirePurchaseOrder">
  + <Descriptor type="label" path="language">
  + <Descriptor type="label" path="maintenanceLog">
  + <Descriptor type="label" path="salesrep">
```

```
  + <Descriptor type="label" path="marketAnalysis">
  + <Descriptor type="label" path="automaticCreditHold">
    <Descriptor type="label" path="requireCreditCheck">
<![CDATA[Require credit check]]>
    </Descriptor>
...*** rest of the descriptors omitted ***
  </MetaData>
  <DomainEntity domainClass="com.mapics.csm.Customer">
    <Key>
      <Property path="company">
        <Value>1</Value>
      </Property>
      <Property path="customer">
        <Value>343400</Value>
      </Property>
    </Key>
    <Property path="company">
      <Value>1</Value>
    </Property>
    <Property path="customerStatus">
      <Value><![CDATA[A]]></Value>
    </Property>
...*** properties omitted ***
    <Property path="requirePurchaseOrder">
      <Value>0</Value>
    </Property>
...*** rest of the properties omitted ***
  </DomainEntity>
</QueryObjectResponse>
```

QueryObjectResponse element

```
<QueryObjectResponse name="queryObject_Customer_Sample"
                     requestedDomainClass="com.mapics.csm.Customer"
                     actionSucceeded="true">
```

The *QueryObjectResponse* element syntax is the same as the *QueryListResponse*. The only difference in this example is that it is using a different object (Customer instead of Item).

MetaData element example Descriptor

```
<Descriptor type="label" path="requireCreditCheck">

- <![CDATA[Require credit check]]>

</Descriptor>
```

A *QueryObject* request is for a card file rather than a list. This is why the meta data returns labels instead of headings. The remaining differences are due to the object and attributes chosen, rather than for any intrinsic difference between *QueryList* and *QueryObject*.

DomainEntity element – multiple key properties

```
<DomainEntity domainClass="com.mapics.csm.Customer">

  <Key>

    <Property path="company">

      <Value>1</Value>

    </Property>

    <Property path="customer">

      <Value>343400</Value>

    </Property>

  </Key>
```

Since this object has two key attributes, we see two property elements between the **Key** elements. Each shows the name of the property (for example, company, customer) and its value (for example,1 and 343400). The value syntax (lack of a CDATA escape sequence) illustrates a numeric-only value for both attributes.

Property elements – alphanumeric, numeric, and boolean

The syntax varies slightly depending on the type of attribute. Attributes can be alphanumeric, numeric, or boolean. A boolean attribute can have only two possibilities, one of which indicates the **true** condition and the other the **false** condition. Whether these values in the database are numeric or alphanumeric is not significant. The XML only communicates if the value is true or false.

Thus, the value of an attribute can be of three types:

- Boolean: will always return a value of 1 (true) or 0 (false). The property below illustrates that the condition of requireCreditCheck is true, that is a credit check is required:

```
<Property path="requirePurchaseOrder">
  <Value>0</Value>
</Property>
```

- Alphanumeric: can be any alphanumeric value. To prevent the possible appearance of XML reserved characters in the value from violating the document syntax, the CDATA escape sequence is used for alphanumeric properties. The property below illustrates that the value of *customerStatus* is A.

```
<Property path="customerStatus">
  <Value><![CDATA[A]]></Value>
</Property>
```

  (Note that even though a particular alphanumeric property may have its possible values limited to a certain set – for example, values in a code file – the property is still considered alphanumeric for the purposes of System-Link providing a CDATA escape sequence.)

- Date: dates are passed as “YYMMDD”. The example property below (not part of the above example response) illustrates that the last credit limit change date was June 19, 1997:

```
<Property path="lastCreditLimitChangeDate">
  <Value><![CDATA[19970619]]></Value>
</Property>
```

- Numeric: can be any numeric data. The example below (not part of the above example response) indicates that the highest credit limit is 875,000:

```
<Property path="highestCreditLimit">
  <Value>8750000</Value>
</Property>
```

  Decimal values are presented in the same way. The property below indicates that the commissionable percent is 0.253. (In this instance it is up to the application to determine whether this value means 25.3% or .253%.)

```
<Property path="commissionablePercent">
<Value>0.253</Value>
</Property>
```

Example 3: Query multiple objects (Manufacturing Purchase Order)

Often we want to query a complex object, such as an order. A complex object is one that has two or more objects related in a header/detail relationship. MOs, COs, POs, and IFM transactions are all examples of complex objects.

In addition, it is often useful to query a collection of objects that are related, although not in a header/detail relationship. Examples of these kinds of interrelationships are:

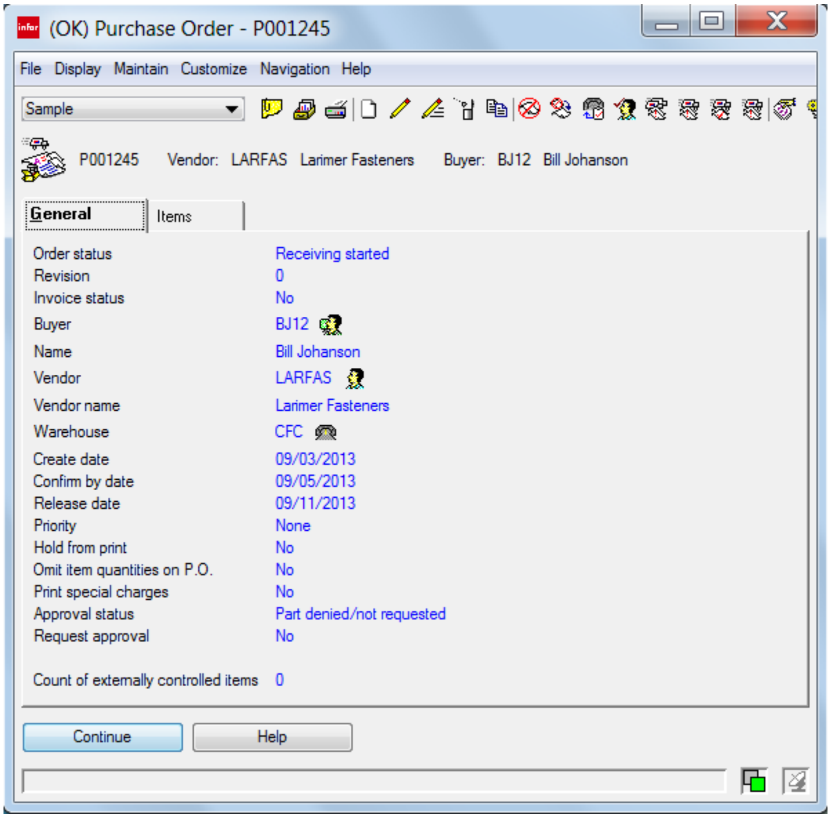
- Customers and COs
- Vendors and POs
- Items and MOs or schedules
- MO components and inventory transactions

System-Link allows you to request multiple objects at one time. For this example, we will request a purchase order with its items, as well as the inventory transactions for the items. This will illustrate the technique for requesting multiple objects (in this case, three):

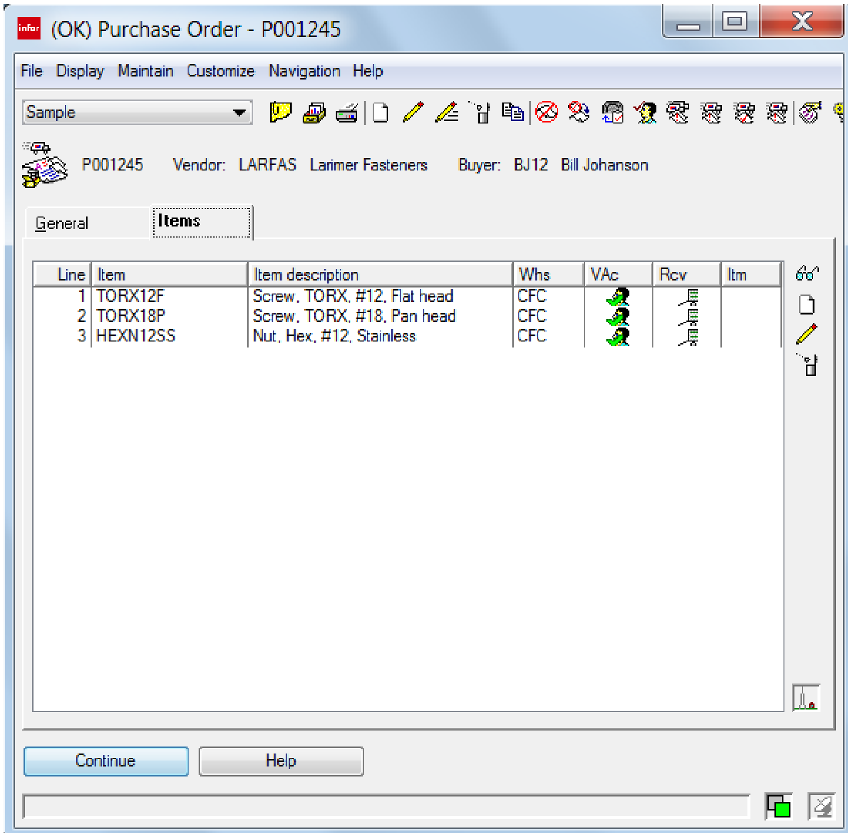
- Purchase Order
- Purchase Order Items
- Inventory Transaction History

The easiest way to start this is to let Power-Link provide the System-Link request. We have created a card file for a PO that contains two cards: one with PO attributes and one list card for the PO Items. We can then ask for the System-Link request.

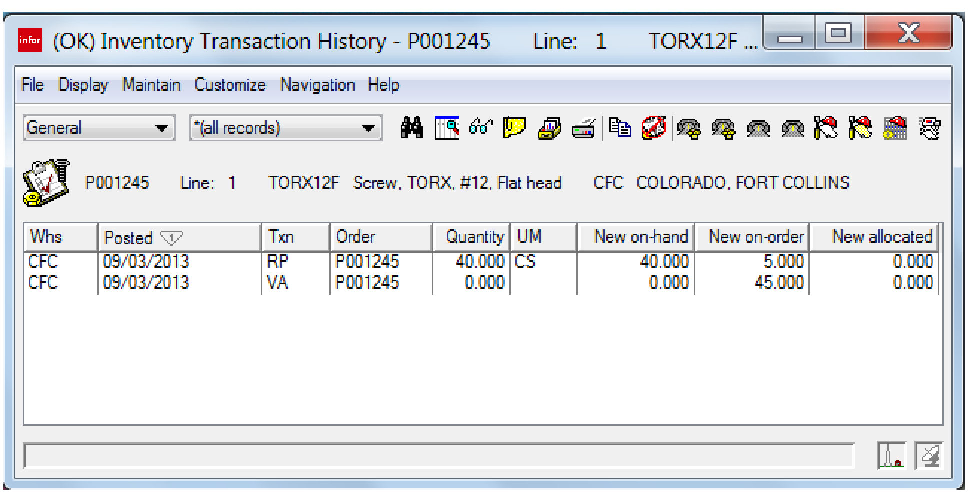
General card: This first card shows eighteen attributes.



Items card: The second card shows three lines, with eight attributes (columns plus order number) each.



Inventory Transaction History: For this view we navigated from the first line on the Items card. The first line has two transactions with nine attributes each. Other lines may also have transactions.



The objective is to have a single XML request that includes:

- A *QueryObject* action for the purchase order and its eighteen attributes
- A *QueryRelationship* action for the PO Item lines and their eight attributes
- A nested *QueryRelationship* action for the transactions for each line, with their nine attributes

System-Link Request

From this card file, we will use the menu option to get the System-Link request. The System-Link request is shown below. This card file does not include inventory transactions:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE System-Link SYSTEM 'SystemLinkRequest.dtd'>
<System-Link>


  <Login userId='&%userid%&' password='&%password%&' maxIdle='900000'
         properties='com.pjx.cas.domain.EnvironmentId=OK,
                     com.pjx.cas.domain.SystemName=usalil2c.infor.com,
                     com.pjx.cas.user.LanguageId=en'/>


  <Request sessionHandle='&%sessionHandle%&' workHandle='*new'
           broker='EJB' maxIdle='1000'>


    <QueryObject name='queryObject_PurchaseOrder_Sample'
                 domainClass='com.mapics.pm.PurchaseOrder'
                 includeMetaData='true'>
          <Pql>
               <![CDATA[SELECT orderStatus,revision,invoiceStatus,
                        buyer,relatedBuyer.buyerName,vendor,
                        vendorName,warehouse,orderCreatedDate,
                        confirmByDate,releaseDate,priority,
                        holdFromPrint,omitItemQuantitiesOnPo,
                        printSpecialCharges,approvalStatus,
                        requestApproval,
     relatedPurchaseOrderExtension.countOfExternallyControlledItems
                        WHERE order = 'P001245']]>
          </Pql>
          <QueryRelationship name='queryRelationship_MyItemsList'
                             relationshipName='relatedPoItems'
                             includeMetaData='true' maxReturned='10'>
               <Pql>
```

```
                    <![CDATA[SELECT order,line,item,itemDescription,
                             warehouse,vendorAcceptanceDerived,
                             receivingStatusDerived,itemStatusDerived
                             WHERE order = 'P001245'
                             ORDER BY order,line]]>

               </Pql>

          </QueryRelationship>

    </QueryObject>

  </Request>


</System-Link>
```

Query Relationship element

The first part of the System-Link request matches what we have seen before. It includes a header section, a session-creation section, the *QueryObject* section to retrieve the purchase order, and the PQL to get the desired attributes. The example above adds the nested QueryRelationship action to the outer QueryObject action. If you only want to see the “main” object, then you see the </QueryObject> tag after the PQL section (therefore closing the QueryObject action “block”). To get the information for the related object(s), the starting tag for QueryRelationship is added, as shown below.

```
<QueryRelationship name='queryRelationship_MyItemsList'
                   relationshipName='relatedPoItems'
                   includeMetaData='true' maxReturned='10'>

    <Pql>

        <![CDATA[SELECT order,line,item,itemDescription,
                 warehouse,vendorAcceptanceDerived,
                 receivingStatusDerived,itemStatusDerived
                 WHERE order = 'P001245'
                 ORDER BY order,line]]>

    </Pql>

</QueryRelationship>
```

This is the System-Link request that will retrieve the list of lines (PO Items) for the PO. You can tell this from the relationship name: relatedPoItems. The relationship name (as opposed to the *QueryRelationship* action name: “queryRelationship_MyItemsList”) is the relationship between the Purchase Orders object and the Purchase Order Items object. This relationship provides the link fields and other information the system needs to retrieve the data.

The *maxReturned* attribute specifies that a maximum of ten component records will be retrieved. Unlike a *QueryList*, though, this is a list query inside an object query. Therefore, you will not be able to retrieve additional records using a *FetchNext* element.

With only one chance to retrieve the data, the temptation would be to make the *maxReturned* value high. This could be a problem, because there is no limit to the number of *QueryRelationships* that can be in a single request. In addition, there is no limit to the number of nesting levels. This means that the amount of data retrieved by a single request can be enormous, and the user would probably not be willing to wait for the time it would take to assemble the data.

The *maxReturned* attribute is the means of controlling the amount of data retrieved. The maximum number of records should be low enough to provide adequate performance given the entire request (that is the total number of *QueryRelationship* actions and their *maxReturned* values). At the same time, it should be large enough to provide value to the user.

The *PQL* defines the PO Item data that will be retrieved:

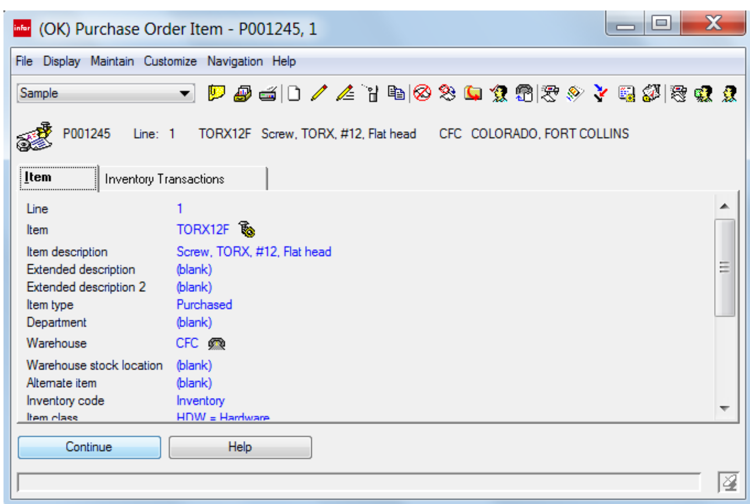
SELECT	The attributes, like columns in a view and possible values used in the header. These will be PO Item attributes.
WHERE	The selection criteria, similar to a subset. We will get all the components for PO P001245.
ORDER BY	The sort order for the component list. Like SQL, by default the data returned will be in ascending order of the named attributes, unless DESC is specified.

Note: Like *QueryList*, *QueryRelationship* actions can also specify custom definitions instead of PQL. We will skip the *QueryDfn* form of *QueryRelationship* for brevity. The *QueryDfn* section is identical to that used with *QueryList*.

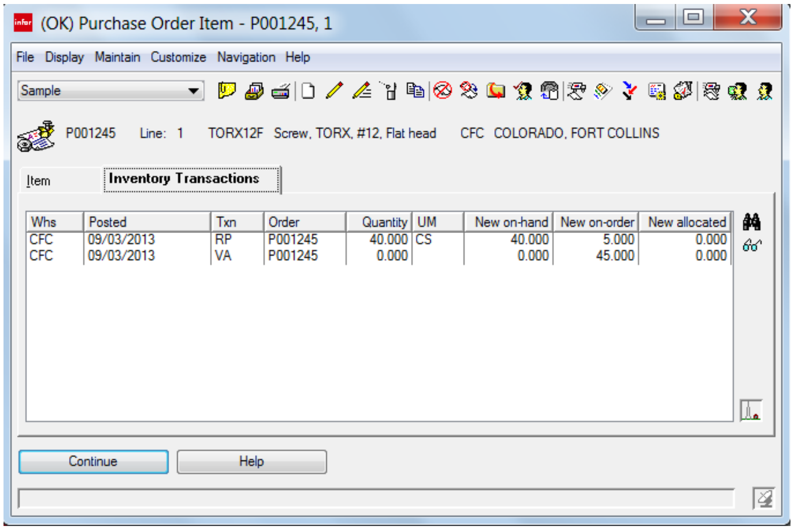
The *QueryRelationship* action closes with the end tag “</QueryRelationship>”.

If we submitted this request we would get eighteen attribute labels (metadata) and values for the PO, and eight attribute headings and values for each of the three lines. But, we still need the inventory transaction information.

To get the System-Link request for the inventory transactions we created a new card file for PO Item. The new card file has two cards: one with attributes (which we do not really need) and one with a list of inventory transactions. This list card is the one that will cause the System-Link request to include a *QueryRelationship*.



To display the PO Item card file we can display the details for one of the lines on the list card in the PO object. We displayed the details for PO item “TORX12F” below. The Inventory Transactions list card looks like this:



The Inventory Transactions list card shows nine attributes (columns in the list). These are the attributes that we want to include in a nested *QueryRelationship* element in our request.

Using the option from the **Navigation** menu, we can display the System-Link request for the PO Item details. The part we care about is the *QueryRelationship* section:

```
<QueryRelationship name='queryRelationship_MyPOInvXactHist'
                   relationshipName='relatedInvTxnHistory'
                   includeMetaData='true' maxReturned='10'>

     <Pql>

          <![CDATA[SELECT item,warehouse,postedDate,transactionCode,
                   orderFormatted,transactionQuantity,
                   unitOfMeasure,newQuantityOnHand,
                   newQuantityOnOrder,newQuantityAllocated
                   WHERE order = 'P001245' AND item = 'TORX12F'
                   ORDER BY postedDate DESC,postedTime DESC]]>

     </Pql>

</QueryRelationship>
```

You can see that this fragment matches the list we have displayed in the list browser, with item added from the heading. Note that the WHERE clause specifies the exact PO Item (TORX12F) that we displayed in the list browser. Making the *QueryRelationship* action apply to all items in the purchase order will be a task that we will address later.

As with all actions, this section starts with a *QueryRelationship* start tag and ends with a *QueryRelationship* end tag. Unique to the *QueryRelationship* action is the way we specify a nested relationship by placing one *QueryRelationship* inside the other, that is, between the beginning and ending tags of the outer action. We will copy this entire section – everything between the starting and ending tags, inclusive – and then go to the earlier fragment to see where to paste it.

Listed below is the previous *QueryRelationship* action for the PO Items.

```
<QueryRelationship name='queryRelationship_MyItemsList'
                   relationshipName='relatedPoItems'
                   includeMetaData='true' maxReturned='10'>

    <Pql>

        <![CDATA[SELECT order,line,item,itemDescription,
                 warehouse,vendorAcceptanceDerived,
                 receivingStatusDerived,itemStatusDerived
                 WHERE order = 'P001245'
                 ORDER BY order,line]]>

    </Pql>

</QueryRelationship>
```

We will paste the *QueryRelationship* action for inventory transactions into the *QueryRelationship* action for PO Items *immediately before* the end element. It should look like the example below, with the nested fragment in bold:

```
<QueryRelationship name='queryRelationship_MyItemsList'
                   relationshipName='relatedPoItems'
                   includeMetaData='true' maxReturned='10'>

    <Pql>

        <![CDATA[SELECT order,line,item,itemDescription,
                 warehouse,vendorAcceptanceDerived,
                 receivingStatusDerived,itemStatusDerived
                 WHERE order = 'P001245'
                 ORDER BY order,line]]>

    </Pql>

     <QueryRelationship name='queryRelationship_MyPOInvXactHist'
                        relationshipName='relatedInvTxnHistory'
                        includeMetaData='true' maxReturned='10'>

          <Pql>

               <![CDATA[SELECT item,warehouse,postedDate,
                        transactionCode,orderFormatted,
                        transactionQuantity,
                        unitOfMeasure,newQuantityOnHand,
                        newQuantityOnOrder,newQuantityAllocated
                        WHERE order = 'P001245' AND item = 'TORX12F'
                        ORDER BY postedDate DESC,postedTime DESC]]>

          </Pql>

     </QueryRelationship>

</QueryRelationship>
```

At this point, we are almost done with our request. The only thing left to fix is the WHERE clause in the nested fragment. Left unchanged we would only see the inventory transactions for item “TORX12F”, which would be repeated in the response for each item. We want to see the applicable transaction records for all the items. To do this we need to understand both the way the nested query relationships are processed as well as the link data.

When System-Link processes the *QueryRelationship* action for PO Item, it will retrieve the first record in the list. Before it looks for a second one, System-Link processes the nested *QueryRelationship* action. This means that System-Link retrieves the entire list of inventory transactions for the first item before looking for the second item. After the second item, System-Link runs the transaction *QueryRelationship* action again. System-Link repeats the process until it has finished the last transaction of the last component (within the bounds of *maxReturned*). Then System-Link proceeds with any subsequent actions in the request document.

The PQL in the nested relationship contains a WHERE clause that identifies the transaction records that will be selected. Right now, these are hard-coded to look for the record we had displayed (the card file) when we asked for the System-Link request from the **Navigation** menu.

Attribute	Value
order	P001245
item	TORX12F

What we want to do is to have these values change depending on the PO Item record. To do this, we use System-Link substitution variables. You can see the names of the attributes in the PO Item *QueryRelationship* PQL that contain the same values as those in the transaction *QueryRelationship* action. Inside PQL, placing ampersands (&) around the attribute name makes that name a System-Link substitution variable. This table shows the association:

Attribute	Value	Substitution variable
order	P001245	&order&
item	TORX12F	&item&

Note: In this case the attribute names are the same for objects on both sides of the relationship. This will often not be the case. The substitution variable should always use the attribute names from the outer or referenced object.

The substitution variables come from the PO item record, but will be used in the transaction *QueryRelationship*. For example, the attribute item in the transaction record is the attribute item in the component record. This is how the link is accomplished.[6] We will substitute the variables containing PO Item values in the WHERE clause of the transaction PQL, thus:

[6] Because the inner QueryRelationship is scoped by the outer action, the substitution variables do not need to be qualified with the name of the outer action. This usage is a special case of the System-Link Value Reference mechanism, which is often used between non-nested actions. Details on this mechanism are given later in this guide.

```
<QueryRelationship name='queryRelationship_MyPOInvXactHist'
                   relationshipName='relatedInvTxnHistory'
                   includeMetaData='true' maxReturned='10'>

     <Pql>

          <![CDATA[SELECT item,warehouse,postedDate,transactionCode,
                   orderFormatted,transactionQuantity,
                   unitOfMeasure,newQuantityOnHand,
                   newQuantityOnOrder,newQuantityAllocated
                   WHERE order = &order& AND item = &item&
                   ORDER BY postedDate DESC,postedTime DESC]]>

     </Pql>

</QueryRelationship>
```

When the first PO Item record is read, both of these attributes will contain values applicable to that record. For example, the value for item will be “TORX12F”. The nested *QueryRelationship* will then be executed, using a WHERE value of “TORX12F” for the item and “P001245” for the order. After the second PO Item record is read, the value of item will be “TORX18P”. The WHERE clause in the nested *QueryRelationship* will then return the transactions for item “TORX18P” and order “P001245”, etc.

Note that for completeness, the *QueryRelationship* for PO Item could also reference the value for attribute order in the outer Purchase Order *QueryObject* action, rather than hardcoding it to “P001245”. We will include this change in the example below as well.

The final request fragment, including the nested query relationship with substitutions, is shown as:

```
<Request sessionHandle='&%sessionHandle%&' workHandle='*new'
         broker='EJB' maxIdle='1000'>


  <QueryObject name='queryObject_PurchaseOrder_Sample'
               domainClass='com.mapics.pm.PurchaseOrder'
               includeMetaData='true'>

        <Pql>

             <![CDATA[SELECT orderStatus,revision,invoiceStatus,
                     buyer,relatedBuyer.buyerName,vendor,
                     vendorName,warehouse,orderCreatedDate,
                     confirmByDate,releaseDate,priority,
                     holdFromPrint,omitItemQuantitiesOnPo,
                     printSpecialCharges,approvalStatus,
                     requestApproval,
    relatedPurchaseOrderExtension.countOfExternallyControlledItems
                     WHERE order = 'P001245']]>

        </Pql>
```

```
        <QueryRelationship name='queryRelationship_MyItemsList'
                           relationshipName='relatedPoItems'
                           includeMetaData='true' maxReturned='10'>

            <Pql>

                <![CDATA[SELECT order,line,item,itemDescription,
                           warehouse,vendorAcceptanceDerived,
                            receivingStatusDerived,itemStatusDerived
                           WHERE order = &order&
                           ORDER BY order,line]]>

            </Pql>

            <QueryRelationship name='queryRelationship_MyPOInvXactHist'
                               relationshipName='relatedInvTxnHistory'
                               includeMetaData='true' maxReturned='10'>

                <Pql>

                   <![CDATA[SELECT item,warehouse,postedDate,
                             transactionCode,orderFormatted,
                             transactionQuantity,unitOfMeasure,
                             newQuantityOnHand,newQuantityOnOrder,
                             newQuantityAllocated
                             WHERE order = &order& AND item = &item&
                             ORDER BY postedDate DESC,postedTime DESC]]>

                </Pql>

            </QueryRelationship>

          </QueryRelationship>

    </QueryObject>

  </Request>
```

System-Link Response

The System-Link response, with nodes collapsed, is shown below. **Note:** As shown by the two lines in bold below, you can see that the purchase order is queried first followed by the lines. The inventory transactions are not visible, because they are nested within the PO Items *QueryRelationshipResponse*.


```
<?xml version="1.0" encoding="UTF-8"?>

<System-Link version="1.0" hostVersion="1.0">

   <LoginResponse actionSucceeded="true">

      <SessionHandle value="-58464f39:140eefbad2e:-7fa2"/>

   </LoginResponse>

   <Response sessionHandle="-58464f39:140eefbad2e:-7fa2"
             workHandle="-58464f39:140eefbad2e:-7fa1"
             systemTimestamp="2013-09-05 12:54:35.392"
             systemTimeZoneOffset="-4:00" hasErrors="false"
             hasWarnings="false">

     <QueryObjectResponse name="queryObject_PurchaseOrder_Sample"
                          requestedDomainClass="com.mapics.pm.PurchaseOrder"
                          actionSucceeded="true">

       + <MetaData>

       + <DomainEntity domainClass="com.mapics.pm.PurchaseOrder">

       + <QueryRelationshipResponse name="queryRelationship_MyItemsList"
                                    relationshipName="relatedPoItems"
                                    actionSucceeded="true"
                                    moreResults="false">

     </QueryObjectResponse>

   </Response>

</System-Link>
```


Below you see the same *Response* section, only with the PO Items node expanded (shown in bold). Notice that the *QueryRelationshipResponse* for PO Items results in a list of three *DomainEntity* elements, that is, three records meaning three items as we saw in the card. After each one, you see a query relationship for transactions, indicating a list of transactions for each item. (Actually, you have to expand the node to see whether there are any transactions. The *QueryRelationshipResponses* may only return metadata.)


```
<Response sessionHandle="-58464f39:140eefbad2e:-7fa2"
          workHandle="-58464f39:140eefbad2e:-7fa1"
          systemTimestamp="2013-09-05 12:54:35.392"
          systemTimeZoneOffset="-4:00" hasErrors="false"
          hasWarnings="false">
```

```
    <QueryObjectResponse name="queryObject_PurchaseOrder_Sample"
                         requestedDomainClass="com.mapics.pm.PurchaseOrder"
                         actionSucceeded="true">

      + <MetaData>

      + <DomainEntity domainClass="com.mapics.pm.PurchaseOrder">

        <QueryRelationshipResponse name="queryRelationship_MyItemsList"
                                   relationshipName="relatedPoItems"
                                   actionSucceeded="true"
                                   moreResults="false">

          + <MetaData>

          + <DomainEntity domainClass="com.mapics.pm.PoItem">

          + <QueryRelationshipResponse
                            name="queryRelationship_MyPOInvXactHist"
                            relationshipName="relatedInvTxnHistory"
                            actionSucceeded="true" moreResults="false">

          + <DomainEntity domainClass="com.mapics.pm.PoItem">

          + <QueryRelationshipResponse
                            name="queryRelationship_MyPOInvXactHist"
                            relationshipName="relatedInvTxnHistory"
                            actionSucceeded="true" moreResults="false">

          + <DomainEntity domainClass="com.mapics.pm.PoItem">

          + <QueryRelationshipResponse
                            name="queryRelationship_MyPOInvXactHist"
                            relationshipName="relatedInvTxnHistory"
                            actionSucceeded="true" moreResults="false">

        </QueryRelationshipResponse>

    </QueryObjectResponse>

</Response>
```

Here we have expanded the first attribute of the first item record to show that it is for PO Item "order P001245, line 1":

```
<Response sessionHandle="-58464f39:140eefbad2e:-7fa2"
          workHandle="-58464f39:140eefbad2e:-7fa1"
          systemTimestamp="2013-09-05 12:54:35.392"
          systemTimeZoneOffset="-4:00" hasErrors="false"
          hasWarnings="false">

    <QueryObjectResponse name="queryObject_PurchaseOrder_Sample"
                         requestedDomainClass="com.mapics.pm.PurchaseOrder"
                         actionSucceeded="true">

      + <MetaData>
```

```
+ <DomainEntity domainClass="com.mapics.pm.PurchaseOrder">
  <QueryRelationshipResponse name="queryRelationship_MyItemsList
                             relationshipName="relatedPoItems"
                             actionSucceeded="true"
                             moreResults="false">
    + <MetaData>
      <DomainEntity domainClass="com.mapics.pm.PoItem">
          <Key>
              <Property path="order">
                  <Value>
                      <![CDATA[ P001245 ]]>
                  </Value>
              </Property>
              <Property path="line">
                  <Value>1</Value>
              </Property>
          </Key>
        + <Property path="order">
        + <Property path="line">
        + <Property path="item">
        + <Property path="itemDescription">
        + <Property path="warehouse">
        + <Property path="vendorAcceptanceDerived">
        + <Property path="receivingStatusDerived">
        + <Property path="itemStatusDerived">
      </DomainEntity>
    + <QueryRelationshipResponse
                            name="queryRelationship_MyPOInvXactHist"
                            relationshipName="relatedInvTxnHistory"
                            actionSucceeded="true" moreResults="false">
    + <DomainEntity domainClass="com.mapics.pm.PoItem">
    + <QueryRelationshipResponse
                            name="queryRelationship_MyPOInvXactHist"
                            relationshipName="relatedInvTxnHistory"
                            actionSucceeded="true" moreResults="false">
```

```
        + <DomainEntity domainClass="com.mapics.pm.PoItem">

        + <QueryRelationshipResponse
                                name="queryRelationship_MyPOInvXactHist"
                                relationshipName="relatedInvTxnHistory"
                                actionSucceeded="true" moreResults="false">

      </QueryRelationshipResponse>

  </QueryObjectResponse>

</Response>
```

Drilling into the response further and continuing to expand the nodes, we can see that the transaction query for the first line has returned two transactions.

```
<QueryRelationshipResponse name="queryRelationship_MyItemsList"
                           relationshipName="relatedPoItems"
                           actionSucceeded="true" moreResults="false">

  + <MetaData>

    <DomainEntity domainClass="com.mapics.pm.PoItem">

        <Key>

            <Property path="order">

                <Value>

                     <![CDATA[ P001245 ]]>

                </Value>

            </Property>

            <Property path="line">

                <Value>1</Value>

            </Property>

        </Key>

      + <Property path="order">

      + <Property path="line">

      + <Property path="item">

      + <Property path="itemDescription">

      + <Property path="warehouse">

      + <Property path="vendorAcceptanceDerived">

      + <Property path="receivingStatusDerived">

      + <Property path="itemStatusDerived">

    </DomainEntity>
```

```
    <QueryRelationshipResponse name="queryRelationship_MyPOInvXactHist"
                               relationshipName="relatedInvTxnHistory"
                               actionSucceeded="true" moreResults="false">

      + <MetaData>

      + <DomainEntity
                  domainClass="com.mapics.mm.InventoryTransactionHistory">

      + <DomainEntity
                  domainClass="com.mapics.mm.InventoryTransactionHistory">

    </QueryRelationshipResponse>

  + <DomainEntity domainClass="com.mapics.pm.PoItem">

  + <QueryRelationshipResponse name="queryRelationship_MyPOInvXactHist"
                               relationshipName="relatedInvTxnHistory"
                               actionSucceeded="true" moreResults="false">

  + <DomainEntity domainClass="com.mapics.pm.PoItem">

  + <QueryRelationshipResponse name="queryRelationship_MyPOInvXactHist"
                               relationshipName="relatedInvTxnHistory"
                               actionSucceeded="true" moreResults="false">

</QueryRelationshipResponse>
```

The first transaction record for the line has a transaction code of RP and a quantity of 40.000 cases.

```
<QueryRelationshipResponse name="queryRelationship_MyItemsList"
                           relationshipName="relatedPoItems"
                           actionSucceeded="true" moreResults="false">

  + <MetaData>

    <DomainEntity domainClass="com.mapics.pm.PoItem">

        <Key>

            <Property path="order">

                <Value>

                    <![CDATA[ P001245 ]]>

                </Value>

            </Property>

            <Property path="line">

                <Value>1</Value>

            </Property>

        </Key>

      + <Property path="order">
```

```
    + <Property path="line">

    + <Property path="item">

    + <Property path="itemDescription">

    + <Property path="warehouse">

    + <Property path="vendorAcceptanceDerived">

    + <Property path="receivingStatusDerived">

    + <Property path="itemStatusDerived">

  </DomainEntity>

  <QueryRelationshipResponse name="queryRelationship_MyPOInvXactHist"
                             relationshipName="relatedInvTxnHistory"
                             actionSucceeded="true" moreResults="false">

    + <MetaData>

      <DomainEntity
                   domainClass="com.mapics.mm.InventoryTransactionHistory">

        + <Key>

        + <Property path="item">

        + <Property path="warehouse">

          <Property path="postedDate">

              <Value>20130903</Value>

          </Property>

          <Property path="transactionCode">

              <Value>

                  <![CDATA[ RP ]]>

              </Value>

          </Property>

        + <Property path="orderFormatted">

          <Property path="transactionQuantity">

              <Value>40.000</Value>

          </Property>

          <Property path="unitOfMeasure">

              <Value>

                  <![CDATA[ CS ]]>

              </Value>
```

```
            </Property>
          + <Property path="newQuantityOnHand">
          + <Property path="newQuantityOnOrder">
          + <Property path="newQuantityAllocated">
        </DomainEntity>
      + <DomainEntity domainClass="com.mapics.mm.InventoryTransactionHistory">
    </QueryRelationshipResponse>
  + <DomainEntity domainClass="com.mapics.pm.PoItem">
  + <QueryRelationshipResponse name="queryRelationship_MyPOInvXactHist"
                               relationshipName="relatedInvTxnHistory"
                               actionSucceeded="true" moreResults="false">
  + <DomainEntity domainClass="com.mapics.pm.PoItem">
  + <QueryRelationshipResponse name="queryRelationship_MyPOInvXactHist"
                               relationshipName="relatedInvTxnHistory"
                               actionSucceeded="true" moreResults="false">
</QueryRelationshipResponse>
```

If we expanded the entire request we would see all the transactions for all the lines, as well as the attributes for each line and for the purchase order itself.

Collapsing and expanding nodes is simply an Internet browser (user interface) convenience. The System-Link response will send all the data (all nodes expanded) without any special programmatic controls.

Example 4: Create or copy a simple object (Warehouse)

Up to now, we have looked at ways to get data out of your system. Now we will look at ways to put data into it. The first maintenance example is for a warehouse. We will create a warehouse called WH1 with these properties:

Attribute name	Value
Warehouse number	WH1
Description	Southeast Shipping Center
Type	True (Controlled)
Site	RON
Shipping calendar	01

The create action is shown below. Login elements, request elements, and all empty and default zero value warehouse properties have been omitted for clarity.

```
<Create name="createObject_Warehouse"
        domainClass="com.mapics.mm.Warehouse">

    <DomainEntity>

        <Key>

            <Property path="warehouse">

                <Value><![CDATA[WH1]]></Value>

            </Property>

        </Key>

        <Property path="site">

            <Value><![CDATA[RON]]></Value>

        </Property>

        <Property path="description">

            <Value><![CDATA[Southeast Shipping Center]]></Value>

        </Property>

        <Property path="securityGroup">

            <Value><![CDATA[9999]]></Value>

        </Property>

        <Property path="warehouseType">

            <Value><![CDATA[1]]></Value>

        </Property>

        <Property path="relatedPlanningInformation.extractRequisitions">
```

```
            <Value><![CDATA[3]]></Value>
        </Property>
        <Property path="relatedMRPPlanningControl.itemsToInclude">
            <Value><![CDATA[1]]></Value>
        </Property>
        <Property path="shippingCalendar">
            <Value><![CDATA[01]]></Value>
        </Property>
        <Property path="relatedMRPPlanningControl.warehouseReschedule">
            <Value><![CDATA[1]]></Value>
        </Property>
        <Property path="relatedPlanningInformation.safetyStockLeadTime">
            <Value><![CDATA[1]]></Value>
        </Property>
        <Property
            path="relatedMRPPlanningControl.contractRequiredForAutoRelease">
            <Value><![CDATA[4]]></Value>
        </Property>
        <Property path="relatedMRPPlanningControl.dateInterval">
            <Value><![CDATA[5]]></Value>
        </Property>
        <Property path="relatedMRPPlanningControl.itemsSelected">
            <Value><![CDATA[1]]></Value>
        </Property>
        <Property path="primaryPlanningWarehouse">
            <Value><![CDATA[*N]]></Value>
        </Property>
        <Property path="backflushCode">
            <Value><![CDATA[0]]></Value>
        </Property>
    </DomainEntity>
</Create>
```

Create and DomainEntity elements

As you would expect, the create action begins with the <Create> start tag. For our example, the <Create> start tag is followed by the <DomainEntity> start tag. (Other elements, like *ReasonCode*, *SourceObject* – for a copy, and *ApplyTemplate* could appear before the *DomainEntity* element. For full details, consult the *System-Link Reference Guide*.) Similarly, the </DomainEntity> end tag is followed by the </Create> end tag. The *Create* action describes each warehouse object that is to be created. You can create multiple objects in one document, but they must each have their own *Create* action.

```
<Create name="createObject_Warehouse"
        domainClass="com.mapics.mm.Warehouse">

    <DomainEntity>
```

We have given the *Create* action the descriptive name "createObject_Warehouse", which happens to match the default generated by the Power-Link request generator. Every name of an action must be unique in its enclosing document. If you create multiple records at once, each one must therefore have its own unique name. We will see how this can be used when we create warehouse addresses.

The *domainClass* tells System-Link that we are creating a warehouse record.

ReasonCode element

When reason tracking is turned on for an object a reason code must be entered when creating, updating, or deleting the object. The *ReasonCode* element is placed in-between the <Create> (<Update> or <Delete>) and <DomainEntity> start tags. The <ReasonCode> start tag is followed by the reason code value and the </ReasonCode> end tag. For more details on how the *ReasonCode* value is specified, consult the *System-Link Reference Guide*.

Other elements

The rest of the request in the *Create* action should be familiar. The **Key**, **Property,** and **Value** elements work in exactly the same way in maintenance as they do in inquiry.

Using a template

You can automatically populate fields by using the *ApplyTemplate* element to reference a template. The example below creates a warehouse record using the *ApplyTemplate* element, shown in bold. Since templates are custom definitions like views, subsets, sorts, cards, and cardfiles, *ApplyTemplate* works in almost the same way as *HeaderDfnKey*, *CardFileDfnKey*, etc. The only differences are:

- A value of "(none)" can be given to *ApplyTemplate* if no template is to be used. (This allows the Power-Link request generator to create a placeholder *ApplyTemplate* action in all *Create* and *Update* actions.)
- Neither *clientClass* nor *classMnemonic* attributes are required. If not provided, System-Link uses the *domainClass* of the outer *Create* action.

Here is how the previous example would look with template Sample applied to the *Create* action:

```
<Create name="createObject_Warehouse"
        domainClass="com.mapics.mm.Warehouse">

    <ApplyTemplate <![CDATA[Sample]]></ApplyTemplate>

    <DomainEntity>
```

When specifying a template, System-Link reads the template values before the property values in the *DomainEntity*. Thus, if the *DomainEntity* element includes information for fields that are also in the template, the *DomainEntity* values will prevail. This allows you to override template values if you wish.

Copy

Copying an object is like creating with a template except that the maintainable field values come from a source object. To indicate this we use the *SourceObject* element. Most *SourceObject* sections will not specify the object class, because it will be the same class as that defined in the outer *Create* element.[7] All the *SourceObject* element has to do is supply the key values of the object to be copied. In the example snippet below, we are creating warehouse "WH1" as a copy of "WH2":

```
<Create name="createObject_Warehouse"
        domainClass="com.mapics.mm.Warehouse">

    <SourceObject>

       <Key>

           <Property path='warehouse'>

               <Value><![CDATA[WH2]]></Value>

           </Property>

       </Key>

    </SourceObject>

    <DomainEntity>

        <Key>

            <Property path="warehouse">

                <Value><![CDATA[WH1]]></Value>

            </Property>

        </Key>

        <Property path="site">
```

[7] There are a very few exceptions to this rule; for example, Customer Orders being copied to Standing Orders. In these instances, the SourceObject element will specify the class for the source object.

```
            <Value><![CDATA[RON]]></Value>
        </Property>
...
```

Note: It is possible to use both *SourceObject* and *ApplyTemplate*. In this case, *ApplyTemplate* should reference a copy template.

Response

Regardless of the use of templates, source objects, and overrides, in the end we get a new "WH1" warehouse record. When the record is created, the response includes the Create action name and the new key. In the example below the Create action name is "createObject_Warehouse" and the key is "WH1":

```
<?xml version="1.0" encoding="UTF-8"?>
<System-Link version="1.0" hostVersion="1.0">
    <LoginResponse actionSucceeded="true">
        <SessionHandle value="-5dd0fa08:14102cccae9:-7faa"/>
    </LoginResponse>
    <Response sessionHandle="-5dd0fa08:14102cccae9:-7faa"
              workHandle="-5dd0fa08:14102cccae9:-7fa9"
              systemTimestamp="2013-09-09 09:00:15.716"
              systemTimeZoneOffset="-4:00" hasErrors="false"
              hasWarnings="false">
        <CreateResponse name="createObject_Warehouse"
                        requestedDomainClass="com.mapics.mm.Warehouse"
                        actionSucceeded="true">
            <Key>
                <Property path="warehouse">
                    <Value><![CDATA[WH1]]></Value>
                </Property>
            </Key>
        </CreateResponse>
    </Response>
</System-Link>
```

Example 5: Create multiple simple objects together (Warehouse with Address)

Often you need to create multiple objects at the same time. For example, we want to create a warehouse with an address:

Warehouse:

Attribute name	Value
Number	WH1
Description	Southeast Shipping Center
Type	True (Controlled)
Site	RON
Shipping calendar	P1

Address:

Attribute name	Value
Name	North Georgia
Address line 1	1013 Hillcrest Road
City	Duluth
State	GA

The XML to create the warehouse was shown in Example 4: Create or copy a simple object (Warehouse). The XML to create an address looks like this:

```
<Create name="createObject_WarehouseAddress"
        domainClass="com.mapics.pm.WarehouseAddress">

    <DomainEntity>

        <Key>

            <Property path="warehouse">

                <Value><![CDATA[WH1]]></Value>

            </Property>

            <Property path="addressId">

                <Value><![CDATA[955]]></Value>

            </Property>

        </Key>

        <Property path="address1">

            <Value><![CDATA[1013 Hillcrest Road]]></Value>
```

```
        </Property>
        <Property path="address4City">
            <Value><![CDATA[Duluth]]></Value>
        </Property>
        <Property path="country">
            <Value><![CDATA[USA]]></Value>
        </Property>
        <Property path="name">
            <Value><![CDATA[North Georgia]]></Value>
        </Property>
        <Property path="state">
            <Value><![CDATA[GA]]></Value>
        </Property>
    </DomainEntity>
</Create>
```

As you can see the *Create* action for the address looks very similar to the *Create* action for the warehouse. Both start and end with <Create> and <DomainEntity> tags, with properties (both key and regular) in the middle.

One single System-Link document can be used to create both objects. The warehouse would need to come first, since one of the address creation edits verifies the existence of the warehouse.

The ValueRef element

The value of the warehouse ID in the address record is equal to the warehouse ID of the warehouse (for example "WH1").

```
<Value><![CDATA[WH1]]></Value>
```

These two values must be the same, so that the new address will be associated with the new warehouse. Instead of typing in the value, "WH1", we can use the *ValueRef* element to tell the property of the address object to use the same value as that in the property of the warehouse object.

```
<ValueRef><![CDATA[(createObject_Warehouse).warehouse]]></ValueRef>
```

In this example, <ValueRef> tags replace <Value> tags, and "(createObject_Warehouse).warehouse" replaces "WH1":

Value	Description
(createObject_Warehouse)	The name of the earlier create action surrounded by parentheses.

Value	Description
warehouse	The property of the create action containing the value to be used.

The *ValueRef* element insures that the value of the warehouse ID in the warehouse *Create* action will automatically be used in the address *Create* action. The *ValueRef* element can be used in this same way for any number of subsequent actions.

This technique is especially useful when the value to be reused is unknown by the XML programmer. For example, an order header may have a system-assigned order number that must be reused for the creation of line item objects. The *ValueRef* element provides a way to tell the line item *Create* actions to use the value from the order number property.

The illustration below shows the warehouse create action followed by the address *Create* action. Note that the *ValueRef* element that gets the value "WH1" from the warehouse property of the "createObject_Warehouse" action. You can see why it is important that all the action names within a work area be unique.

```
<Request sessionHandle="*current" workHandle="*new" broker="EJB"
         maxIdle="1000">

    <Create name="createObject_Warehouse"
            domainClass="com.mapics.mm.Warehouse">

        <DomainEntity>

            <Key>

                <Property path="warehouse">

                    <Value><![CDATA[WH1]]></Value>

                </Property>

            </Key>

            <Property path="site">

                <Value><![CDATA[RON]]></Value>

            </Property>

            ... (rest of properties omitted for clarity) ...

        </DomainEntity>

    </Create>

    <Create name="createObject_WarehouseAddress"
            domainClass="com.mapics.pm.WarehouseAddress">

        <DomainEntity>

            <Key>

                <Property path="warehouse">
```

```
                    <Value><![CDATA[WH1]]></Value>
                </Property>
                <Property path="addressId">
                    <Value><![CDATA[955]]></Value>
                </Property>
            </Key>
            <Property path="address1">
                <Value><![CDATA[1013 Hillcrest Road]]></Value>
            </Property>
            ... (rest of properties omitted for clarity) ...
        </DomainEntity>
    </Create>
</Request>
```

Example 6: Create a complex object (Purchase Order)

A complex object contains multiple objects in a header/detail relationship. COs, POs, MOs, IFM transactions, and engineering bills of material are all examples of complex objects. In most cases, System-Link handles the creation of a complex object by using separate *Create* actions for the various components: root object (header) first and child objects subsequent. This request structure is similar to what we previously saw with *QueryObject* and *QueryRelationship*, except that none of the *Create* actions are nested within each other.

StartTransactionGroup element

In most instances, we want all components of a complex object creation to be treated atomically: all are successfully created or none are created. To enable this behavior, we add the *StartTransactionGroup* element.

```
<StartTransactionGroup/>
```

The *StartTransactionGroup* element has neither attributes nor any subelements. Adding this element within a *Request* section causes all maintenance actions between the *StartTransactionGroup* element and the </Request> end tag to be grouped into the same transaction, and to be processed once the end of the *Request* section is reached.

Note that *StartTransactionGroup* does not apply to inquiry actions; these will still be processed as System-Link encounters them in the document. When writing a request using *StartTransactionGroup*, one generally should avoid adding inquiry actions that depend upon the outcome of a maintenance action in that request document.

Request

This example shows the creation of a purchase order with one line and one release:

Header

(attribute)	(value)
Number	System-assigned
Buyer	100
Warehouse	CFC
Vendor	123

Line item

(attribute)	(value)
PO number	Same as header
Line number	System-assigned
Warehouse	CFC

Item	TORX12F
Quantity	100.0
Unit of Measure	Case
Blanket	Yes

Release

(attribute)	(value)
PO number	Same as header
PO line	Same as line item
PO release	System-assigned
Due date	September 27, 2013
Quantity	5

Notes:

1. The *Create* action for the header includes the *Key* element with no properties. This tells System-Link that the missing properties are intentional. System-Link will pass the request to the lower IDF layers, where the key (that is, the PO number) will be assigned.
2. The line item has a two-part key: PO number and line number. We use a *ValueRef* element to tell the system to insert the value of the PO number from the first *Create* action. (The first *Create* action must have *retainResult=‘true’* for this to work.) The second part of the key (the line number) will be assigned, just like the PO number was in the first *Create* action.
3. The release has a three part key with the first two properties using *ValueRef* elements to the PO and line item. As with the PO *Create* action, the PO Item *Create* action must also have *retainResult='true'*.
4. The technique described above also applies to the release (three-part key).
5. All dates within System-Link are given in YYYYMMDD format.
6. The blanketItem property for the line uses value "1" for "true". Boolean properties in System-Link requests can have values of 1/0 or true/false. Boolean values in responses will always use 1/0. (Boolean attributes, like "retainResult", will always use true/false.)

As usual, the *Login* and root (System-Link) elements have been omitted for clarity.

```
<Request sessionHandle="*current" workHandle="*new"
         broker="EJB" maxIdle="1000">

    <StartTransactionGroup/>

    <Create name="createObject_PurchaseOrder"
            domainClass="com.mapics.pm.PurchaseOrder"
            retainResult="true">

        <DomainEntity>
```

```
            <Key>
            </Key>
            <Property path="warehouse">
                <Value><![CDATA[CFC]]></Value>
            </Property>
            <Property path="billToId">
                <Value><![CDATA[142]]></Value>
            </Property>
            <Property path="buyer">
                <Value><![CDATA[100]]></Value>
            </Property>
            <Property path="vendor">
                <Value><![CDATA[123]]></Value>
            </Property>
            <Property path="shipToId">
                <Value><![CDATA[142]]></Value>
            </Property>
        </DomainEntity>
    </Create>
    <Create name="createObject_PurchaseOrderItem"
            domainClass="com.mapics.pm.PoItem"
            retainResult="true">
        <DomainEntity>
            <Key>
                <Property path="order">
                    <ValueRef>
                            <![CDATA[(createObject_PurchaseOrder).order]]>
                    </ValueRef>
                </Property>
            </Key>
            <Property path="item">
                <Value><![CDATA[TORX12F]]></Value>
            </Property>
```

```
            <Property path="orderQuantityRequested">
                <Value>100.000</Value>
            </Property>
            <Property path="warehouse">
                <Value><![CDATA[CFC]]></Value>
            </Property>
            <Property path="unitPriceRequested">
                <Value>60.0000</Value>
            </Property>
            <Property path="blanketItem">
                <Value>1</Value>
            </Property>
            <Property path="orderUm">
                <Value><![CDATA[CS]]></Value>
            </Property>
        </DomainEntity>
    </Create>
    <Create name="createObject_PurchaseOrderItemRelease"
            domainClass="com.mapics.pm.PoItemRelease">
        <DomainEntity>
            <Key>
                <Property path="order">
                    <ValueRef>
                            <![CDATA[(createObject_PurchaseOrder).order]]>
                    </ValueRef>
                </Property>
                <Property path="line">
                    <ValueRef>
                           <![CDATA[(createObject_PurchaseOrderItem).line]]>
                    </ValueRef>
                </Property>
            </Key>
            <Property path="dueToStockDate">
                <Value><![CDATA[20130927]]></Value>
```

```
            </Property>

            <Property path="releaseQuantity">

                <Value>5.000</Value>

            </Property>

        </DomainEntity>

    </Create>

</Request>
```

Result

The result shows each *Create* action name and the key values for the record that was created. *Login* and root elements have been omitted for clarity.

```
<Response sessionHandle="-5dd0fa08:14102cccae9:-7e80"
          workHandle="-5dd0fa08:14102cccae9:-7e7f"
          systemTimestamp="2013-09-09 11:39:18.280"
          systemTimeZoneOffset="-4:00" hasErrors="false"
          hasWarnings="false">

    <CreateResponse name="createObject_PurchaseOrder"
                    requestedDomainClass="com.mapics.pm.PurchaseOrder"
                    actionSucceeded="true">

        <Key>

            <Property path="order">

                <Value><![CDATA[P001262]]></Value>

            </Property>

        </Key>

    </CreateResponse>

    <CreateResponse name="createObject_PurchaseOrderItem"
                    requestedDomainClass="com.mapics.pm.PoItem"
                    actionSucceeded="true">

        <Key>

            <Property path="order">

                <Value><![CDATA[P001262]]></Value>

            </Property>

            <Property path="line">

                <Value>1</Value>

            </Property>
```

```
        </Key>
    </CreateResponse>
    <CreateResponse name="createObject_PurchaseOrderItemRelease"
                    requestedDomainClass="com.mapics.pm.PoItemRelease"
                    actionSucceeded="true">
        <Key>
            <Property path="order">
                <Value><![CDATA[P001262]]></Value>
            </Property>
            <Property path="line">
                <Value>1</Value>
            </Property>
            <Property path="release">
                <Value>1</Value>
            </Property>
        </Key>
    </CreateResponse>
</Response>
```

Example 7: ‘Create’ an action (PO receipt)

Sometimes the create operation is not for an object but for an action performed on an object. In this case, we want to receive items against a PO.

Property	Value
PO Number	P001262
PO Line	1
PO Release	1
Quantity received to stock	2.0 cases

Note: The warehouse associated with this example PO is uncontrolled. If the warehouse were controlled, additional properties of stockLocation, fifoDate, and batchLot would be supported.

The System-Link action is a creation of the receipt, as shown:

```
<Create name="newReceiveItem" domainClass="com.mapics.mm.ReceiveItem">
    <ApplyTemplate clientClass="com.mapics.mm.ReceiveItem">Stock receipts
    only</ApplyTemplate>
    <DomainEntity>
        <Key>
            <Property path="orderNumber">
                <Value><![CDATA[P001262]]></Value>
            </Property>
            <Property path="line">
                <Value>1</Value>
            </Property>
            <Property path="release">
                <Value>1</Value>
            </Property>
        </Key>
        <Property path="quantityReceivedForStock">
            <Value>2.0</Value>
        </Property>
        <Property path="receiptUm">
            <Value><![CDATA[CS]]></Value>
        </Property>
```

```
    </DomainEntity>

</Create>
```

The response includes the name of the create action and the key values:

Name	newReceiveItem	
Key	PO	P001262
	Line	1
	Rel	1

The *Login*, *Response*, and root elements have been omitted for clarity.

```
<CreateResponse name="newReceiveItem"
                requestedDomainClass="com.mapics.mm.ReceiveItem"
                actionSucceeded="true">

    <Key>

        <Property path="orderNumber">

           <Value><![CDATA[P001262]]></Value>

        </Property>

        <Property path="line">

           <Value>1</Value>

        </Property>

        <Property path="release">

           <Value>1</Value>

        </Property>

    </Key>

</CreateResponse>
```

Example 8: Update (PO line)

The *Update* element is very similar to the *Create* element, both:

- Require a name for the action
- Require the domain class to be identified
- Can have many of the same nested elements: *ApplyTemplate*, *DomainEntity*, etc.

In this example, we are making the these changes:

Property	Value	Details
PO number	P001263	(key field)
Line	1	(key field)
Quantity requested	200	
Unit price requested	57	There is no currency shown because it is not being changed.

The illustration below shows the *Update* action. *Login*, *Request*, and root elements have been omitted for clarity.

```
<Update name="updateObject_PurchaseOrderItem"
        domainClass="com.mapics.pm.PoItem">

    <DomainEntity>

        <Key>

            <Property path="order">

                <Value><![CDATA[P001263]]></Value>

            </Property>

            <Property path="line">

                <Value><![CDATA[1]]></Value>

            </Property>

        </Key>

        <Property path="orderQuantityRequested">

            <Value><![CDATA[200.000]]></Value>

        </Property>

        <Property path="unitPriceRequested">

            <Value><![CDATA[57.0000]]></Value>

        </Property>

    </DomainEntity>
```

```
</Update>
```

A successful update will result in an *UpdateResponse* with no subelements and *actionSucceeded="true"*:

```
<Response sessionHandle="-5dd0fa08:14102cccae9:-7de0"
          workHandle="-5dd0fa08:14102cccae9:-7ddf"
          systemTimestamp="2013-09-09 13:41:16.036"
          systemTimeZoneOffset="-4:00" hasErrors="false"
          hasWarnings="false">

    <UpdateResponse name="updateObject_PurchaseOrderItem"
                    requestedDomainClass="com.mapics.pm.PoItem"
                    actionSucceeded="true"/>

</Response>
```

An unsuccessful *Update* action will result in the *UpdateResponse* element containing one or more *Exception* messages inside the <UpdateResponse> start and end tags, as well as a "false" value for *actionSucceeded* and a "true" value for *hasErrors* on the *Request* element.

Example 9: Delete (Warehouse)

The *Delete* action is very straightforward – all you have to do is tell System-Link the name of the object and its key. In this example, the object is warehouse and the key is "WH3". *Login*, *Request,* and root elements have been omitted for clarity.

```
<Delete name="oldWarehouse" domainClass="com.mapics.mm.Warehouse">

    <DomainEntity>

        <Key>

            <Property path="warehouse">

                <Value><![CDATA[WH3]]></Value>

            </Property>

        </Key>

    </DomainEntity>

</Delete>
```

Similar to the *Update* action, a successful *Delete* action will result in a *DeleteResponse* with no subelements and *actionSucceeded="true"*:

```
<Response sessionHandle="-5dd0fa08:14102cccae9:-7db9"
          workHandle="-5dd0fa08:14102cccae9:-7db8"
          systemTimestamp="2013-09-09 13:48:30.845"
          systemTimeZoneOffset="-4:00" hasErrors="false"
          hasWarnings="false">

    <DeleteResponse name="oldWarehouse"
                    requestedDomainClass="com.mapics.mm.Warehouse"
                    actionSucceeded="true"/>

</Response>
```

As with the update action an unsuccessful *Delete* action will result in the *DeleteResponse* element containing one or more *Exception* messages inside the <DeleteResponse> start and end tags, as well as a "false" value for *actionSucceeded* and a "true" value for *hasErrors* on the *Request* element.

Appendix A Supported objects

A

Information regarding the domain classes supported by System-Link can be obtained dynamically via the Utility action *ListDomainClassInfo*.

Example: To display a System-Link response listing all domain classes supported, this action would be used:

```
<ListDomainClassInfo name='listAllClasses' domainClass='*all'/>
```

For further information on obtaining domain class lists, as well as determining each supported class actions and properties, see “*ListDomainClassInfo”* in the *IDF System-Link Reference Guide*.

Note: If you wish to use System-Link with Integrator objects or IDF primary objects that have been extended with Integrator, you must have Enterprise Integrator installed. These objects will not be usable through System-Link with Standard Integrator. In the case of extended IDF primary objects, this means that the entire object (not just its extension) will be unavailable through System-Link without installing Enterprise Integrator.