eaSDK: Building Custom Jobs

eaSDK is a member of the eaSuite™ product line

V3.4
Document ID: SDGN-06-3.4-02
Data Published: 2.6.2003

Table of Contents

Preface

About Customer Self-Service and eaSuite™

eaSuite

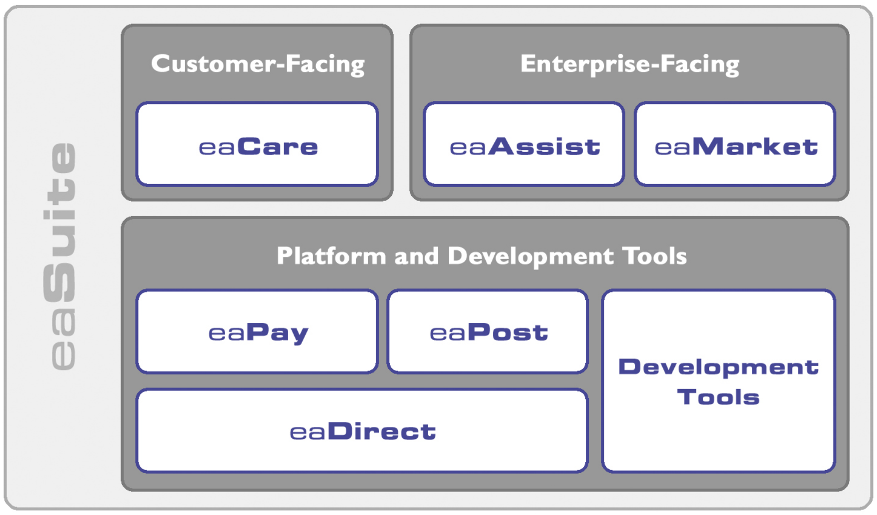
edocs has developed the industry's most comprehensive software and services for deploying Customer Self-Service solutions. eaSuite™ combines electronic presentment and payment (EPP), order management, knowledge management, personalization and application integration technologies to create an integrated, natural starting point for all customer service issues. eaSuite's unique architecture leverages and preserves existing infrastructure and data, and offers unparalleled scalability for the most demanding applications. With deployments across the healthcare, financial services, energy, retail, and communications industries, and the public sector, eaSuite powers some of the world's largest and most demanding customer self-service applications. eaSuite is a standards-based, feature rich, and highly scalable platform, that delivers the lowest total cost of ownership of any self-service solution available.

eaSuite is designed to support how organizations approach designing and deploying Customer Self-Service applications:

Customer-Facing Solutions present customers with the sophisticated functionality to meet customers' self-service needs. eaSuite offers a full set of capabilities to enable the range of business and consumer customer service activities, along with the flexibility to completely customize the solution to meet vertical industry and specific company requirements.

Enterprise-Facing Solutions empower employees within an organization and external partners to leverage the edocs platform to facilitate self-service and to support assisted service. Customer service representatives (CSRs), sales agents, account managers, marketing managers, broker-dealers and channel partners all play a role in delivering customer service, creating content, accessing information and performing activities for the benefit of customers.

Platform and Development Tools are designed to meet the rigorous infrastructure demands of the most technologically advanced organizations. These components of the eaSuite power edocs solutions with the functionality and development tools necessary to make account data available, and to create the customer- and enterprise-facing applications that enable customer self-service.



eaAssist

eaAssist™ reduces interaction costs and increases customer satisfaction by enabling enterprise agents – customer service representatives (CSRs), sales agents, broker-dealers and others – to efficiently access critical account data and service-related information to effectively service customers. Through its browser interface designed especially for the enterprise agent, eaAssist enables agents to take advantage of customer-facing online capabilities to provide better service by more efficiently resolving customer account inquiries at the point of customer contact.

eaMarket

eaMarket™ is the personalization, campaign and content management solution that enables organizations to increase revenue and improve customer satisfaction by weaving personalized marketing and customer service messages throughout the Customer Self-Service experience. The transactional account data that provides the foundation for a Customer Self-Service solution – such as transaction activity, service or usage charges, current task and prior service history – bring valuable insight into customers and can help optimize personalized marketing and customer service campaigns. eaMarket leverages that data to present relevant marketing and customer service messages to customers.

eaDirect

eaDirect™ is the core infrastructure of enterprise Customer Self-Service solutions for organizations large and small with special emphasis on meeting the needs of organizations with large numbers of customers, high data volumes and extensive integration with systems and business processes across the enterprise. Organizations use eaDirect with its data access layer, composition engine, and security, enrollment and logging framework to power complex Customer Self-Service applications.

eaPay

eaPay™ is the electronic payment solution that decreases payment processing costs, accelerates receivables and improves operational efficiency. eaPay is a complete payment scheduling and warehousing system with real-time and batch connections to payment gateways for Automated Clearing House (ACH) and credit card payments, and payments via various payment processing service providers.

eaPost

eaPost® is the account content distribution system that handles all the complexities of enrollment, authentication and secure distribution of summary account information to any endpoint, while also bringing customers back the organization's Website to manage and control their self-service experience.

Development Tools

eaSuite Development Tools™ are visual development applications that provide intuitive graphical user interface (GUI) environments for designing and developing Customer Self-Service solutions. The Development Tools encompass data management, workflow authoring, rules management and accounts receivable integration, as well as a full Software Developers Kit for custom application development.

About This Guide

The edocs Software Developers Kit (eaSDK) allows developers to write custom code against edocs applications. This SDK is intended for edocs system integrator partners, senior developers with an edocs client company, and edocs Professional Services representatives.

eaSDK Building Custom Jobs is intended for software developers defining and adding a custom job type to the Command Center.

The eaSDK assumes you have:

- Completed a Statement Mastering Plan

- Installed and configured eaDirect and the sample application eaSample

This SDK assumes in-depth understanding of and practical experience with:

- eaDirect system architecture, installation, deployment, application design, and administration
- Java 2 Enterprise Edition (J2EE), Enterprise JavaBeans (EJBs), servlets, and JSPs
- Packaging and deploying J2EE applications for WebLogic 6.1 or WebSphere 4.0
- Directory services including the Java Naming Directory Interface (JNDI) and the Lightweight Directory Access Protocol (LDAP)
- HTML and XML, web server administration, and web browsers

Related Documentation

Javadoc for the SDK, Online Help for DefTool and Composer, and a PDF version of this guide are also available.

Online	How to Access
Javadoc	SDK Javadoc is available on the eaDirect SDK CD-ROM.
Help	Select Help > Help Topics in DefTool or Composer.
PDF	A PDF of this guide is available on the eaDirect SDK CD-ROM.

This guide is part of the eaDirect documentation set. For more information about implementing your eaDirect application, see one of the following guides:

Print Document	Description
eaDirect Installation and Configuration Guide	How to install eaDirect and configure it in a distributed environment.

Print Document	Description
eaDirect User's Guide	How to design and create an eaDirect application and use the eaDirect DefTool and Composer.
eaDirect Production Guide	How to set up and run a live eaDirect application in a J2EE environment.

If You Need Help

Technical support is available to customers who have valid maintenance and support contracts with edocs. Technical support engineers can help you install, configure, and maintain your edocs application.

To reach the U.S. Service Center, located in Natick, MA (Monday through Friday 8:00am to 8:00pm EST):

- Telephone: 508.652.8400
- Toll Free: 877.336.3362
- E-support: support.edocs.com (This requires a one-time online registration)
- E-mail: support@edocs.com

When you report a problem, please be prepared to provide us the following information:

- What is your name and role in your organization?
- What is your company's name?
- What is your phone number and best times to call you?
- What is your e-mail address?
- In which edocs product did a problem occur?
- What is your Operating System version?
- What were you doing when the problem occurred?

- How did the system respond to the error?
- If the system generated a screen message, please send us that screen message.
- If the system wrote information to a log file, please send us that log file.

If the system crashed or hung, please tell us.

About Custom Job Types 1

This guide describes how to create custom job types that include the Shell Command Task. This task allows you to run an external command script to process the output files from other tasks within the job. You can use this guide to:

- Define a custom job type for the Command Center and create a SQL script, containing job type and task information, to add the new job type.
- View and configure the new job type in the Command Center.

About Jobs and the Shell Command Task

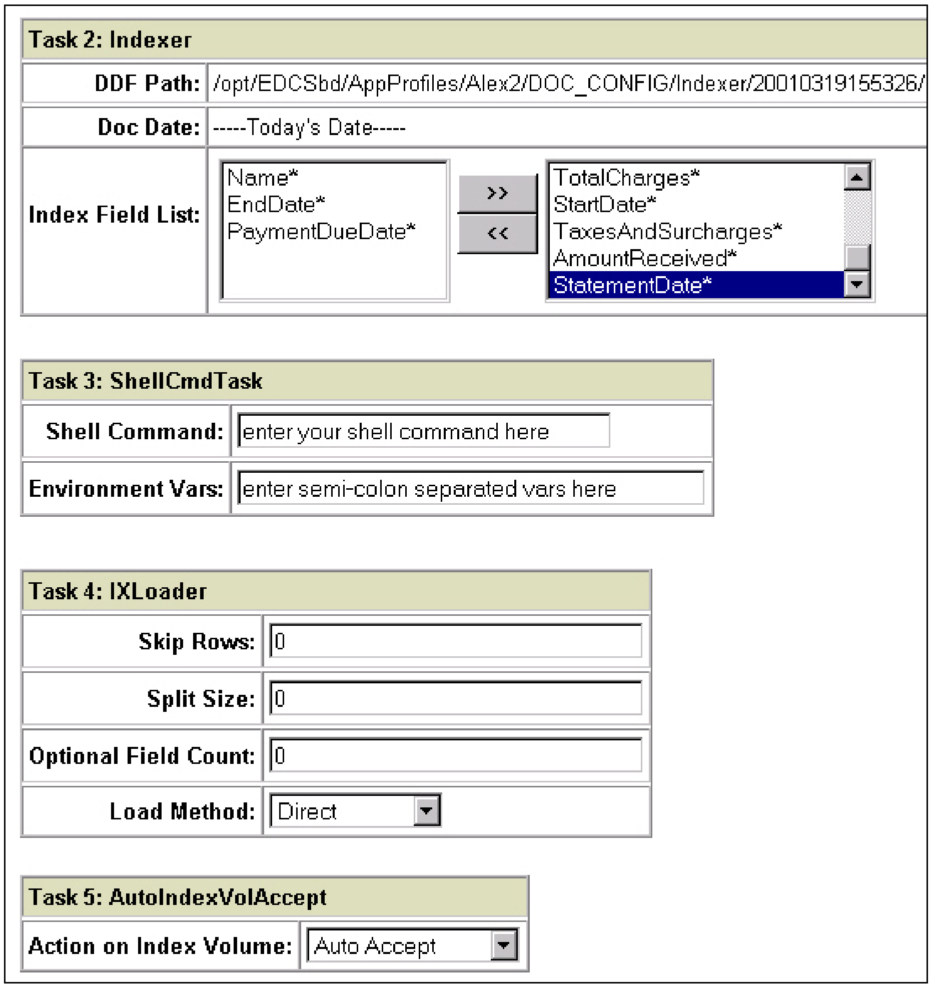
eaDirect has several predefined job types available in its Command Center. Each job is made up of one or more tasks. For complete listing of jobs and tasks, see the *eaDirect Production Guide*.

However, there may be times when you will want to expand these predefined Jobs to fit your needs. For cases like this eaDirect has the ability to define your own **custom Job** Type that you can make up from a combination of the predefined tasks that come with eaDirect and/or your own custom task by defining what is referred to in eaDirect as a **Shell Command Task**.

A Shell Command Task is a way of invoking a shell script, executable, or other program that was written to perform a task specific to your requirements. It enables you to run custom scripts or programs, such as pre- or post- processors as part of a user-defined job. You can create your own Job Type by creating a SQL script that updates the database. Once the database is updated this Job Type becomes available to you via the Command Center. The new Job Type can then be configured, scheduled, and run from within the Command Center.

For example, you could create a new custom job called **`Preprocess`** to run a pre-processor on the input file in an Indexer job. At runtime, this job would insert a custom shell command task **`ShellCmdTask`** between the Scanner and the Indexer tasks. Another use would be to create a job to run a validation engine (sum all amount due, for example) on the output of the Indexer task. At runtime, the **`SumAllDue`** task would be inserted between the Indexer and the IXLoader tasks.

The following illustration shows a new custom Indexer job type in the Command Center for the latter example.



Task 2: Indexer

DDF Path:	/opt/EDCSbd/AppProfiles/Alex2/DOC_CONFIG/Indexer/20010319155326/
Doc Date:	-----Today's Date-----
Index Field List:	Name* EndDate* PaymentDueDate* / TotalCharges* StartDate* TaxesAndSurcharges* AmountReceived* StatementDate*

Task 3: ShellCmdTask

Shell Command:	enter your shell command here
Environment Vars:	enter semi-colon separated vars here

Task 4: IXLoader

Skip Rows:	0
Split Size:	0
Optional Field Count:	0
Load Method:	Direct

Task 5: AutoIndexVolAccept

Action on Index Volume:	Auto Accept

Defining a New Job Type 2

This chapter includes information about:

- Creating the job type script
- Configuring the new job type
- Examples of the job type script

Create the Job Type Script

To create a job type you create a single SQL script to run in the eaDirect database using the Oracle utility *sqlplus*. Within this SQL script you define:

1 The job name

2 The tasks and the order in which they will run

3 The input arguments

The following sections provide a detailed topic description of each part. Each section uses the example of specifying a new job type that is similar to the current Indexer job, except that between scanning for an input file (Scanner Task) and actually indexing the file (Indexer Task) you need to invoke a preprocessor to modify the input file. This is the situation where you need to insert the ShellCmdTask between the other tasks.

Example *sqlplus* script for Oracle

```
DECLARE jtid NUMBER;
BEGIN

  -- Define the job name
  jtid := pwc_job_types.create_job_type ('myIndexer');

  -- Specify the job tasks and their order

  pwc_job_types.create_job_type_task(jtid,'Scanner', 1);

  pwc_job_types.create_job_type_task(jtid,'ShellCmdTask', 2);

  pwc_job_types.create_job_type_task(jtid,'Indexer', 3);

  pwc_job_types.create_job_type_task(jtid,'IXLoader',
4);

  pwc_job_types.create_job_type_task(jtid,
'AutoIndexVolAccept', 5);

  -- Define the tasks input arguments

  pwc_job_types.create_job_type_io(jtid,'ShellCmdTask',
'input params', 'INPUT', 2,'Scanner', 'output file
name', 'OUTPUT', 1);

  pwc_job_types.create_job_type_io(jtid, 'Indexer','data
file name', 'INPUT', 3, 'ShellCmdTask','shell output',
'OUTPUT', 2);

  pwc_job_types.create_job_type_io(jtid, 'Indexer', 'ddn
volume number', 'INPUT', 3,'Scanner', 'ddn volume
number', 'OUTPUT', 1);

  pwc_job_types.create_job_type_io(jtid, 'IXLoader',
'index volume number', 'INPUT', 4,'Scanner', 'ddn volume
number', 'OUTPUT', 1);

  pwc_job_types. create_job_type_io (jtid, 'IXLoader',
'ir file name', 'INPUT', 4, 'Indexer', 'ir file name',
'OUTPUT', 3);

  pwc_job_types.create_job_type_io(jtid,
'AutoIndexVolAccept', 'index volume number', 'INPUT', 5,
'Scanner', 'ddn volume number, 'OUTPUT', 1);


END;
```

Example script for AIX/DB2

To create a DB2 shell command for a custom job in AIX, run the following command:

```
db2 -td@ -vf customjob.sh
```

where **`customjob.sh`** is the name of a shell script customized for your job, platform, and environment. See the example below for a sample script to customize.

```
DROP PROCEDURE db2inst1.tmp_pwc_jtt_sp() @

CREATE PROCEDURE db2inst1.tmp_pwc_jtt_sp()

   LANGUAGE SQL

BEGIN

   DECLARE jtid             INTEGER;

   DECLARE l_job_type_name  VARCHAR(32);

   DECLARE l_task_name      VARCHAR(32);

   DECLARE l_task_order     INTEGER;

   DECLARE l_i_task_name    VARCHAR(32);

   DECLARE l_i_task_io_name VARCHAR(32);

   DECLARE l_i_task_io_type VARCHAR(32);

   DECLARE l_i_task_order   INTEGER;

   DECLARE l_o_task_name    VARCHAR(32);

   DECLARE l_o_task_io_name VARCHAR(32);

   DECLARE l_o_task_io_type VARCHAR(32);

   DECLARE l_o_task_order   INTEGER;


    -- job type with
'Scanner':'ShellCmdTask':'Indexer':'IXLoader':'AutoIndex
VolAccept'

   SET l_job_type_name = 'Custom_Indexer';

   CALL pwc_job_types.create_job_type(jtid,
l_job_type_name);
```

```
   SET l_task_name = 'Scanner';
   SET l_task_order = 1;
   CALL pwc_job_types.create_job_type_task(jtid,
l_task_name, l_task_order);
   SET l_task_name = 'ShellCmdTask';
   SET l_task_order = 2;
   CALL pwc_job_types.create_job_type_task(jtid,
l_task_name, l_task_order);
   SET l_task_name = 'Indexer';
   SET l_task_order = 3;
   CALL pwc_job_types.create_job_type_task(jtid,
l_task_name, l_task_order);
   SET l_task_name = 'IXLoader';
   SET l_task_order = 4;
   CALL pwc_job_types.create_job_type_task(jtid,
l_task_name, l_task_order);
   SET l_task_name = 'AutoIndexVolAccept';
   SET l_task_order = 5;
   CALL pwc_job_types.create_job_type_task(jtid,
l_task_name, l_task_order);

   SET l_i_task_name    = 'ShellCmdTask';
   SET l_i_task_io_name = 'input params';
   SET l_i_task_io_type = 'INPUT';
   SET l_i_task_order   = 2;
   SET l_o_task_name    = 'Scanner';
   SET l_o_task_io_name = 'output file name';
   SET l_o_task_io_type = 'OUTPUT';
   SET l_o_task_order   = 1;
```

```
CALL pwc_job_types.create_job_type_io(jtid,
l_i_task_name, l_i_task_io_name,

l_i_task_io_type, l_i_task_order, l_o_task_name,
l_o_task_io_name, l_o_task_io_type,

l_o_task_order);

   SET l_i_task_name    = 'Indexer';

   SET l_i_task_io_name = 'data file name';

   SET l_i_task_io_type = 'INPUT';

   SET l_i_task_order   = 3;

   SET l_o_task_name    = 'ShellCmdTask';

   SET l_o_task_io_name = 'shell output';

   SET l_o_task_io_type = 'OUTPUT';

   SET l_o_task_order   = 2;

CALL pwc_job_types.create_job_type_io(jtid,
l_i_task_name, l_i_task_io_name,

l_i_task_io_type, l_i_task_order, l_o_task_name,
l_o_task_io_name, l_o_task_io_type,

l_o_task_order);

   SET l_i_task_name    = 'Indexer';

   SET l_i_task_io_name = 'ddn volume number';

   SET l_i_task_io_type = 'INPUT';

   SET l_i_task_order   = 3;

   SET l_o_task_name    = 'Scanner';

   SET l_o_task_io_name = 'ddn volume number';

   SET l_o_task_io_type = 'OUTPUT';

   SET l_o_task_order   = 1;

CALL pwc_job_types.create_job_type_io(jtid,
l_i_task_name, l_i_task_io_name,

l_i_task_io_type, l_i_task_order, l_o_task_name,
l_o_task_io_name, l_o_task_io_type,

l_o_task_order);
```

```
   SET l_i_task_name    = 'IXLoader';
   SET l_i_task_io_name = 'index volume number';
   SET l_i_task_io_type = 'INPUT';
   SET l_i_task_order   = 4;
   SET l_o_task_name    = 'Scanner';
   SET l_o_task_io_name = 'ddn volume number';
   SET l_o_task_io_type = 'OUTPUT';
   SET l_o_task_order   = 1;
CALL pwc_job_types.create_job_type_io(jtid,
l_i_task_name, l_i_task_io_name,
l_i_task_io_type, l_i_task_order, l_o_task_name,
l_o_task_io_name, l_o_task_io_type,
l_o_task_order);
   SET l_i_task_name    = 'IXLoader';
   SET l_i_task_io_name = 'ir file name';
   SET l_i_task_io_type = 'INPUT';
   SET l_i_task_order   = 4;
   SET l_o_task_name    = 'Indexer';
   SET l_o_task_io_name = 'ir file name';
   SET l_o_task_io_type = 'OUTPUT';
   SET l_o_task_order   = 3;
CALL pwc_job_types.create_job_type_io(jtid,
l_i_task_name, l_i_task_io_name,
l_i_task_io_type, l_i_task_order, l_o_task_name,
l_o_task_io_name, l_o_task_io_type,
l_o_task_order);
   SET l_i_task_name    = 'AutoIndexVolAccept';
   SET l_i_task_io_name = 'index volume number';
   SET l_i_task_io_type = 'INPUT';
   SET l_i_task_order   = 5;
   SET l_o_task_name    = 'Scanner';
   SET l_o_task_io_name = 'ddn volume number';
   SET l_o_task_io_type = 'OUTPUT';
   SET l_o_task_order   = 1;
CALL pwc_job_types.create_job_type_io(jtid,
l_i_task_name, l_i_task_io_name,
l_i_task_io_type, l_i_task_order, l_o_task_name,
l_o_task_io_name, l_o_task_io_type,
l_o_task_order);


END @
CALL db2inst1.tmp_pwc_jtt_sp() @
DROP PROCEDURE db2inst1.tmp_pwc_jtt_sp() @
```

Name the Job

The first part of the script is to give your new task a name. The syntax to do this is:

```
jtid := pwc_job_types.create_job_type ('<new_job_name>');
```

In the script, the **`create_job_type`** call defines a unique job type ID (**`jtid`**) for the new *Indexer1* job type.

So if your new job name is *myIndexer*, then the code script will be:

```
jtid := pwc_job_types.create_job_type ('myIndexer');
```

Specify Job Tasks

The next step is to specify what tasks will be part of the new job, and in what order will they execute. The syntax is:

```
pwc_job_types.create_job_type_task(jtid,'<task_name>', n);
```

where *n* equals the order number of the task and *jtid* is the job type id – created with **`pwc_job_types.create_job_type()`** function. The **`create_job_type_task`** call defines the order of the tasks in the job.

In the above example, the plan is to create a new job type based on the current Indexer job type. The tasks included in the Indexer Job are (in their order of execution):

Job	Tasks
Indexer	Scanner
	Indexer
	IXLoader
	AutoIndexVolAccept

If you insert the ShellCmdTask after the Scanner**Error! Bookmark not defined.** Task, it will become task 2, and the others will be incremented by one. The code example is:

```
pwc_job_types.create_job_type_task(jtid,'Scanner', 1);

pwc_job_types.create_job_type_task(jtid,'ShellCmdTask',
2);

pwc_job_types.create_job_type_task(jtid,'Indexer', 3);

pwc_job_types.create_job_type_task(jtid,'IXLoader', 4);

pwc_job_types.create_job_type_task(jtid,
'AutoIndexVolAccept', 5);
```

Define Input Arguments

Each task has input and output arguments, and a particular task may require the output arguments from a previous task to function properly. For example, in the default Indexer job, its Indexer task takes two input arguments from the Scanner Task. In the SQL Script you define which specific input arguments for a task are used from the specific output arguments from another task.

For a list of arguments, see the Javadoc for the *eaSDK 3.0 API Specification.*

To define the input and output parameters, the following is the syntax of the function of the call that uses nine arguments:

```
pwc_job_types.create_job_type_io(jtid,
                                 '<input_task_name>',
                                 '<input_argument>',
                                 'INPUT',
                                 x,
                                '<output_task_name>',
                                '<output_argument>',
                                'OUTPUT',
                                y);
```

where *x* is the order number of the input task and *y* is the order number of the output task. The **`create_job_type_io`** calls define the input values for each job task. It accepts the following parameter values:

- The job type ID (**`jtid`**)
- The task name receiving the input value
- The input parameter name
- The I/O type (**`INPUT`**)
- The order number for the task receiving the input value (defined earlier in the script)
- The previous task name dispensing the output to be used for input
- The output parameter name from the previous task
- The I/O type (**`OUTPUT`**)
- The order number of the task dispensing the output value (defined earlier in the script)

The following breaks down the input arguments used in the above example script:

```
pwc_job_types.create_job_type_io(jtid,
                                'ShellCmdTask',
                                'input params',
                                'INPUT',
                                2,
                                'Scanner',
                                'output file name',
                                'OUTPUT',
                                1);
```

The input argument *input params* for the ShellCmdTask uses the output argument *output file name* from the Scanner task.

```
pwc_job_types.create_job_type_io(jtid,
                                 'Indexer',
                                 'data file name',
                                 'INPUT',
                                 3,
                                 'ShellCmdTask',
                                 'shell output',
                                 'OUTPUT',
                                 2);
pwc_job_types.create_job_type_io(jtid,
                                 'Indexer',
                                 'ddn volume number',
                                 'INPUT',
                                 3,
                                 'Scanner',
                                 'ddn volume number',
                                 'OUTPUT',
                                 1);
```

The input arguments *data file name* and *ddn volume number* for the Indexer task uses the output arguments *shell output* from the ShellCmdTask and *ddn volume number* from the Scanner task respectively.

```
pwc_job_types.create_job_type_io(jtid,
                                 'IXLoader',
                                 'index volume number',
                                 'INPUT',
                                 4,
                                 'Scanner',
                                 'ddn volume number',
                                 'OUTPUT',
                                 1);
```

```
pwc_job_types. create_job_type_io (jtid,
                                    'IXLoader',
                                    'ir file name',

                                    'INPUT',
                                    4,
                                    'Indexer',
                                    'ir file name',
                                    'OUTPUT',
                                    3);
```

The input arguments *index volume number* and *ir file name* for the IXLoader task uses the output arguments *ddn volume number* from the Scanner and *ir file name* from the Indexer respectively.

```
pwc_job_types.create_job_type_io(jtid,
                                  'AutoIndexVolAccept',
                                  'index volume number',
                                  'INPUT',
                                  5,
                                  'Scanner',
                                  'ddn volume number,
                                  'OUTPUT',
                                  1);
```

The input argument *index volume number* for the AutoIndexVolAccept task uses the output argument *ddn volume number* from the Scanner task.

Configuring Your New Job Type

After creating the script, you will need to run it against the Oracle database used by eaDirect (as described in the *Installation and Configuration Guide*). For example, if the script is named *myindexer.sql* and placed in */opt/EDCSbd/db* (the default database location for eaDirect), you could run the following in SQL*Plus:

```
$ sqlplus -s edx_dba/edx@edx.db @ /opt/EDCSbd/db/myindexer.sql
```

The above command presumes you are using the default names for the eaDirect database (edx0) and database administrator/password (edx_dba/edx).

Tip

Before the new job type is available in the Command Center, you have to stop and start Your application server after running the script.

Once the new job type is available to you from the Command Center, you can define the new job using that new job type.

Define the Shell Command Task

If you have included the ShellCmdTask with your new job type, it has 2 input fields to define:

- Shell Command
- Environment variables

The Shell Command field defines the location of the shell script to execute on your system. Note that the user that starts the application server must have read/execute permissions for that location.

The shell command must output, on its standard output, the name of its output file that is the input file to be processed by the next task in the job. If the shell command doesn't output any file name, the job stops as a no-op. If it is successful, the shell command must set its exit code to 0.

If the shell command fails, it must set its exit code to a non-zero value. Additionally, it may output, on its standard error, a message describing the failure. The error message will be logged into the log file by eaDirect. However, any errors within the shell command are not logged and must be handled separately.

For example, the following shell command would be useful after the Scanner task to ensure Windows files have the correct format for UNIX:

```
#!/bin/csh
# Preprocessor to run dos2unix on the input file

dos2unix $SHELL_INPUT $SHELL_INPUT.ux >& /dev/null
if ($status != 0) exit $status          # failure
echo $SHELL_INPUT.ux                    # new input
file
exit 0                                  # success
```

The Environment variables field specifies the environment variables for the shell command. By default, the external command is passed the following environment variables:

- DDN - the name of the application to which the job belongs
- JOB_NAME - the name of the job to which the task is a part of.
- STATUS - the status of the job (has it been started, did it succeed/fail, etc).
- PREVIOUS_STATUS
- SHELL_INPUT - any input from a previous task. The SHELL_INPUT variable is only set if the shell command task is linked with another task in the context of a job. Otherwise it is null.

If your shell command requires any other environment variables, you'll need to specify them in this field.

Another Example of Defining a New Job Type

The following is another example that defines an index job called Indexer2 with the following tasks:

Job	**Tasks**
Indexer2	Scanner
	Indexer
	ShellCmdTask
	IXLoader
	AutoIndexVolAccept

As mentioned in the previous chapter, a reason for this new job type could be to run a validation engine (sum all amount due for example) on the output of the Indexer task. If the amount due exceeds a certain amount, it may require a careful verification of the input data stream as described in the SDK Module: Auditing Data Streams with the Verify API.

For this case you can create the following SQL script:

```
DECLARE jtid NUMBER;
BEGIN
```

```
    jtid := pwc_job_types.create_job_type('Indexer2');

    pwc_job_types.create_job_type_task(jtid, 'Scanner',
1);

    pwc_job_types.create_job_type_task(jtid, 'Indexer',
2);

    pwc_job_types.create_job_type_task(jtid,
'ShellCmdTask', 3);

    pwc_job_types.create_job_type_task(jtid, 'IXLoader',
4);

    pwc_job_types.create_job_type_task(jtid,
'AutoIndexVolAccept', 5);

    pwc_job_types.create_job_type_io(jtid, 'Indexer',
'data file name', 'INPUT', 2, 'Scanner', 'output file
name', 'OUTPUT', 1);

    pwc_job_types.create_job_type_io(jtid, 'Indexer',
'ddn volume number', 'INPUT', 2, 'Scanner', 'ddn volume
number', 'OUTPUT', 1);

    pwc_job_types.create_job_type_io(jtid,
'ShellCmdTask', 'input params', 'INPUT', 3, 'Indexer',
'ir file name', 'OUTPUT', 2);

    pwc_job_types.create_job_type_io(jtid, 'IXLoader',
'index volume number', 'INPUT', 4, 'Scanner', 'ddn
volume number', 'OUTPUT', 1);

    pwc_job_types.create_job_type_io(jtid, 'IXLoader',
'ir file name', 'INPUT', 4, 'Indexer', 'ir file name',
'OUTPUT', 2);

    pwc_job_types.create_job_type_io(jtid,
'AutoIndexVolAccept', 'index volume number', 'INPUT', 5,
'Scanner', 'ddn volume number', 'OUTPUT', 1);

END;
```

Index