

Developer's Guide

Netscape Application Server:
Process Automation Edition

Version 4.0

Netscape Communications Corporation ("Netscape") and its licensors retain all ownership rights to the software programs offered by Netscape (referred to herein as "Software") and related documentation. Use of the Software and related documentation is governed by the license agreement accompanying the Software and applicable copyright law.

Your right to copy this documentation is limited by copyright law. Making unauthorized copies, adaptations, or compilation works is prohibited and constitutes a punishable violation of the law. Netscape may revise this documentation from time to time without notice.

THIS DOCUMENTATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. IN NO EVENT SHALL NETSCAPE BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY KIND ARISING FROM ANY ERROR IN THIS DOCUMENTATION, INCLUDING WITHOUT LIMITATION ANY LOSS OR INTERRUPTION OF BUSINESS, PROFITS, USE, OR DATA.

The Software and documentation are copyright ©1999 Netscape Communications Corporation. All rights reserved.

Netscape, Netscape Navigator, Netscape Certificate Server, Netscape DevEdge, Netscape FastTrack Server, Netscape ONE, SuiteSpot, and the Netscape N and Ship's Wheel logos are registered trademarks of Netscape Communications Corporation in the United States and other countries. Other Netscape logos, product names, and service names are also trademarks of Netscape Communications Corporation, which may be registered in other countries. Other product and brand names are trademarks of their respective owners.

The downloading, exporting, or reexporting of Netscape software or any underlying information or technology must be in full compliance with all United States and other applicable laws and regulations. Any provision of Netscape software or documentation to the U.S. Government is with restricted rights as described in the license agreement accompanying Netscape software.



Recycled and Recyclable Paper

Version 4.0

Part Number 151-07663-00

Copyright 1999 Netscape Communications Corp. All rights reserved.

Printed in the United States of America. 00 99 98 5 4 3 2 1

Netscape Communications Corporation, 501 East Middlefield Road, Mountain View, CA 94043

Contents

Introduction	23
About This Guide	23
Assumptions	24
Conventions Used in This Guide	24
Viewing Documentation Online	25
Chapter 1 Introduction to Process Builder	27
About Processes and Process Builder	27
What is a Process?	28
What is Process Builder?	28
Who Should Use Process Builder?	28
What is a Process Instance?	28
About Applications	29
Sample Applications	29
Creating an Application	29
Deploying an Application	30
Starting Process Builder	31
The preferences.ini File	31
Specifying a cluster	31
Specifying a corporate directory	32
Adding or Changing Local Application Folders	32
Starting Process Builder on Windows NT	33
Starting Process Builder on Solaris	33
Using Process Builder	34
Process Builder Applications	36
Application Tree View	38
Process Map	39
Map Palette	39
Activities Tab	40

Documentation Tab	41
Messages Window	41
Main Toolbar	42
Menu Commands	43
Application	43
Edit	45
Insert	45
Format	45
Window	46
Help	47
HTML Page Editor Toolbar	47
Chapter 2 Planning an Application	49
Planning Overview	50
Planning the Process Map	50
Entry Points	51
User Activities	51
Automated Activities	52
Custom Activities	52
Subprocesses	52
Decision Points	53
Parallel Processing	53
Exit Points	53
Transitions	54
Notifications	54
Exceptions	54
Planning Assignments	54
Planning Groups and Roles	55
Planning for Monitoring	56
Planning Delegations	57
Determining the Data Requirements	57
File Attachments	58
Planning Forms	58
Planning Access to Forms	59

Planning Custom Scripts	59
Planning Searches	60
Planning Deployment	60
Chapter 3 Creating an Application	61
Application Creation Overview	62
Getting Information from the Administrator	63
Creating a New Application	63
The New Application Dialog Box	64
Setting Application Properties	65
The Application Properties Dialog Box	66
Using Sample Applications	68
Applications and the Corporate Directory	69
Setting Your Corporate Directory	70
Deleting an Application	71
Chapter 4 Designing a Process Map	73
Drawing the Process Map	74
Saving a Process Map to a File	74
Adding Items with the Palette	75
Deleting Items	77
Entry Points	77
User Activities	78
Setting Activity Expirations	79
Setting Activity Assignments	81
Using Parallel Approval	82
Parallel Approval Completion Script	84
Automated Activities	84
Subprocesses	86
Connecting the Parent and Child Process	90
Custom Activities	91
Using a Custom Activity	91
Custom Activity Inspector Window	91
Inspector Window After Setting a Custom Activity	94
Adding a Custom Palette	95

Exception Manager	97
Default Exception Manager	98
Creating an Exception Manager	98
Exception Manager Properties	99
Decision Points	100
Split-Join (Parallel Processing)	101
Properties of a Parallel Process	102
Adding a Parallel Process	102
Notifications	104
Notification Properties	104
Built-in Email Notification Scripts	105
Exit Points	105
Transitions	106
Types of Transitions	106
Adding a Transition	107
Transition Properties	107
Setting the Property for a Virtual Transition	107
Setting the Property for a Conditional Transition	108
Example Using a True/False Field	108
Chapter 5 Defining Groups and Roles	111
Groups and Roles Overview	112
Default Groups and Roles	113
Creating Groups and Roles	114
The Create a New Role or Group Dialog Box	115
Types of Groups and Roles	115
Adding a New Group or Role	116
The Application Group Dialog Box	117
The Corporate Group Dialog Box	121
The Dynamic Group Dialog Box	125
The Field Role Dialog Box	127

Prioritizing Groups and Roles	128
Deleting Groups and Roles	130
Chapter 6 Defining Data Fields	131
Data Field Overview	132
Creating a Data Field	132
The Create a New Data Field Dialog Box	133
Creating a Custom Data Field	134
Creating a Predefined Data Field	135
Adding a Data Field	135
Setting Field Properties	136
Custom Data Fields with Predefined Class IDs	137
CheckBox	137
Computed	138
Date	138
DateTime	139
Digital Signature	139
File Attachment	140
Usage Tips for File Attachments	140
File Attachment Properties	141
Java Applet	142
Java Bean	144
Password	144
Radio Buttons	145
Select List	146
TextArea	147
TextField	148
URL	149
UserPicker Widget	150
Custom Data Fields with Your Own Class ID	151
Predefined Data Fields	151
Address	152
Name	153
Telephone	154

Deleting Data Fields	155
Setting Up the Content Store	156
The Content Store Inspector Window	156
Troubleshooting the Content Store	157
Chapter 7 Designing Forms	159
Planning Forms	160
Creating Forms	160
Modifying Forms	165
Using Process Builder's Form Editor	165
Using an External Editor	165
Using the HTML Page Editor Toolbar	166
Using the Edit, Insert and Format Menus	167
Edit	167
Insert HTML Element	167
Format	168
Using Right-Mouse-Button Menu Commands	168
Changing Field Properties for a Form	169
Hints for Setting Field Properties	170
Accessing the Data Dictionary	171
Using Scripts to Validate User Input	171
Adding a Banner to Forms	171
Setting Access to Forms	172
Forms for Assignees	173
Forms for Monitoring the Process	174
Forms for the Administrator	174
Setting Access to an Entry Point	174
Chapter 8 Using Scripts	177
Overview of Scripts	178
Kinds of Scripts	178
When to Use Scripts	180
About Writing Scripts	181
Predefined Scripts	181
Assignment Scripts	181

Completion Scripts	184
Email Notification Scripts	184
defaultNotificationHeader()	185
defaultNotificationSubject()	185
emailByDN(DN)	185
emailById(userId)	186
emailOfAssignees()	186
emailOfCreator()	186
emailOfRole(role)	186
Initialization and Shutdown Scripts	187
Creating Scripts	187
The Script Editor Window	189
Setting a Script as a Template	190
Using Client-side Scripts	191
Chapter 9 Setting Up Searching	195
Types of Searching	196
Global Searching	197
Application-Specific Searching	197
Enabling Searching	198
Allowing a Group to Search	198
Setting Up Forms	198
Allowing Searching for Fields	200
Chapter 10 Deploying an Application	201
Before You Deploy	202
Set up and Configure PAE	202
Deploy Subprocesses First	202
Save the Process Map, If Desired	202
Fix Application Errors	203
Steps for Deploying an Application	203
The Deploy Application Dialog Box	206
Revising a Deployed Application	207
Summary of Allowed Revisions	208
Changes to Activities and Transitions	208

Changes to Data Elements	209
Changes to Forms, Scripts, and Content Store	209
Deployed Applications Compared with Local Copies	210
Using a Backup of a Local Application	210
Saving a Local Application to Another Name	211
Redeploying an Application	212
Chapter 11 The Data Sheet Application	213
Data Sheet Application Overview	214
Data Sheet Process Map	214
Data Sheet Walkthrough	216
The Data Sheet Entry Point	216
The Title Field	217
Add Art Activity	217
Assignment Script	217
Expiration Setter Script	217
The Forms	218
The Automated Activity	219
The Approve Pricing (Product Manager) Activity	220
Assignment Script	220
Forms	221
Notification	221
The VP Approval Decision Point	222
The Approve Price (VP) Activity	222
Assignment Script	223
Forms	223
Exit Points	223
Groups and Roles	224
Data Dictionary	224
Forms	226
Script Dictionary	229
The buildDS Script	229
The lookupCode Script	230
The buildDataSheet Script	231

The computeTitle Script	234
Content Store	235
Finished Data Sheet Example	235
The description.txt File	235
The image.gif File	236
The printer.html Template File	237
The Finished Data Sheet	239
Configuring the Data Sheet Application	241
Configuration Hints	241
How Users Access the Data Sheet	241
Process Express and Netscape Application Server	242
Adding an Email Attribute for a User	242
Using File Attachments and Content Stores	243
Step 1: Administer the Enterprise Server	244
Step 2: Edit the Access Control List	244
Step 3: Assign the Content Store's Style	245
Step 4: Set the Values for the Application's Content Store	245
Chapter 12 The Office Setup Application	247
Office Setup Application Overview	248
Office Setup Process Map	248
Office Setup Walkthrough	249
Start Entry Point	249
Assign Office and Specify Computer Work Item	251
Basic Setup Split	253
Set Up Phone Work Item	254
Install Network Connection Work Item	256
Order Computer Work Item	258
Install Computer Work Item	260
Basic Setup Join	262
CheckSetup Work Item	262
Setup Complete Exit Point	264

The Office Setup Groups	264
Data Dictionary	265
Form Dictionary	267
Script Dictionary	269
Completion Scripts	269
setRequesterField	269
verifySetup	270
Toolkit Scripts	271
buildExitNotification	271
Embedded Client-Side Script	272
Customizing the Appearance of the Forms	273
Configuring the Office Setup Application	274

Chapter 13 The Loan Management and Credit History Applications

275

Loan Management Application Overview	276
Credit History Application Overview	276
Loan Management Process Map	276
Credit History Process Map	277
Loan Management and Credit History Walkthrough	278
New Loan Request Entry Point	279
Verification Work Item	280
Check Credit History Subprocess	281
Check Credit History Subprocess Properties	281
Data Mapping	283
Completion Script	284
The Subprocess Failed Exception Manager	284
Credit History Application as a Subprocess	284
Check Authorization Work Item	285
LookUp Credit History Automated Activity	287
Canceled Exit Point	287
OK Exit Point	288
Meeting Work Item (Parent Process)	289
Loan Refused Exit Point	291

Loan Approved Exit Point	291
Groups and Roles	292
Data Dictionary	292
Loan Management Script Dictionary	293
storeCreditInformation Completion Script	293
CustomerId Toolkit Script	295
Credit History Script Dictionary	295
LookUpCreditHistory Automation Script	296
Configuring the Loan Management Application	296
Configuring the Credit History Application	297
Chapter 14 The Insurance Claim Processing Application	299
Application Overview	300
Process Map	301
Entry Point	302
Activities	302
Custom Activities	302
Look Up Details	302
Update Log	303
Log Denial	303
Decision Point	303
Parallel Process	303
Exit Points	304
Notification	304
Application Walkthrough	304
The Entry Point (Enter Policy No.)	304
Custom Activity (Look Up Details)	305
Activity (Policy Details)	305
Activity (Re-enter Policy No.)	305
Activity (Enter Claim Details)	305
Activity (Confirmation)	306
Activity (Approver)	306
Activity (Clarification)	306
Activity (Inform Customer)	307

Parallel Activity	307
Activity (Send Check)	307
Custom Activity (Update Log)	307
Groups and Roles	307
Group and Role Priorities	308
Data Dictionary	309
Forms	311
Entry.html	311
Resubmit.html	312
Details.html	313
EnterClaim.html	314
Confirmation.html	315
Approver.html	316
Clarify.html	317
InfCustomer.html	318
SendCheck.html	319
ClaimApproved.html	319
ClaimDenied.html	319
ExpirationExit.html	319
Script Dictionary	320
Expiration Scripts	320
Script at Policy Details Activity	320
Script at Enter Claim Details Activity	320
Script at Confirmation	321
Script at Resubmit Policy No.	321
Completion Scripts	321
Completion Script at Policy Details Activity	322
Completion Script at Enter Claim Details Activity	322
Required Files	322
The database.xml File	322
The Policy.log File	324
The Banner Image	325
The Background Image	325

Configuring the Insurance Claim Processing Application	325
Custom Activity Code	326
The LogPerformer Activity	326
The LogdenialPerformer Activity	326
The LookupPerformer Activity	327
Code Walkthrough for LookupPerformer.java	327
Definitions and Packages	327
The init, perform, and destroy Methods	328
The GetPolicy Method	331
Chapter 15 Advanced Techniques for Scripting	333
Introduction	333
Getting Information about the Current Process	334
Getting and Setting Data Field Values	335
Getting Data Field Values in Decision Point and Automation Script Transitions	336
Getting Information about Users and their Attributes	336
Finding Users and Accessing their Attributes	336
Modifying User Attributes	337
Verifying an Array of User DNs	338
Adding and Deleting Users	339
Accessing the Content Store	339
Example of Accessing a Stored Item	339
Storing Files in the Content Store	341
Logging Error and Informational Messages	342
Verifying Form Input	342
Verifying Form Input with Client-Side JavaScript	343
Event Handler Example	344
onSubmitForm Example	346
Verifying Form Data in Completion Scripts	347
Initializing and Shutting Down Applications	348
Debugging Hints	349
Undefined Items	349
Adding Helpful Messages to Script Failure Dialog Boxes	350

Displaying the Progress of a Script	350
Testing Expiration Setter and Handler Scripts	351
Sample Scripts	351
Assignment Script	352
Expiration Setter Script	353
Expiration Handler Script	354
Completion Script	354
Automation Script	355
Chapter 16 Scripting with EJB Components	359
Calling EJB Components from JavaScript	360
A Sample Script	361
Handling Exceptions	363
Data Conversion Issues	364
Chapter 17 Writing Custom Activities	365
Introduction	366
Comparison to Automated Activities	366
Usage Overview	366
Implementing ISimpleWorkPerformer	367
Methods of ISimpleWorkPerformer	367
The init() method	367
The perform() method	368
The destroy() method	369
Sample Java Class	369
Creating HelloWorldPerformer.java	370
Writing the XML Description File	372
File Format	373
WORKPERFORMER Tag	374
ENVIRONMENT Section	374
INPUT Section	375
OUTPUT Section	376
PARAMETER Tag	377
DESIGN Section	378
Sample XML Description File	378

Packaging a Custom Activity	380
Adding a Custom Activity to the Process Map	382
Adding a Custom Activity from a Custom Palette	382
Adding a Custom Activity without Using a Custom Palette	385
Working with a Custom Activity	386
Implementation Tips	387
Avoid Instance Data	387
Use Consistent Data Types	387
Avoid Non-default Constructors	387
When to Use a Custom Activity	388
Chapter 18 Writing Custom Fields	389
Introduction	389
Why Use a Custom Field?	390
Functional View of a Custom Field	390
Steps for Creating a Custom Field	391
Defining Field Properties in a JSB File	391
JSB_DESCRIPTOR Tag	392
JSB_PROPERTY Tag	393
JSB_PROPERTY Attributes	394
Required Data Field Properties	395
Writing the Java Classes	396
Consider the Design Issues	396
Consider Your Data and Data Sources	396
Design a Data Manager	397
Design a Thread-safe Class	399
Use an Entity Key	400
Implement IPresentationElement and IDataElement	400
Displaying a Work Item	402
Initiating a Process Instance	403
Completing a Work Item	404
Accessing a Custom Field from a Script	405

Packaging a Custom Field	405
Adding a Custom Field to an Application	406
Method Reference	408
IPresentationElement Interface	409
display()	409
update()	410
IDataElement Interface	410
create()	411
store()	411
load()	412
archive()	413
BasicCustomField Class	413
loadDataElementProperties()	414
IPMElement Interface	415
getName()	415
getPrettyName()	416
Appendix A JavaScript API Reference	417
ProcessInstance	418
getConclusion()	418
getCreationDate()	418
getCreatorDN()	419
getCreatorUser()	419
getData (fieldName)	419
getEntityKey(fieldName)	420
getEntryNodeName()	420
getExitNodeName()	420
getInstanceId ()	421
getPriority()	421
getRoleDN(roleName)	421
getRoleUser(roleName)	422
getTitle()	422
setEntityKey(fieldName, value)	423
setData (fieldName, fieldValue)	423

setPriority(value)	424
setRoleByDN(roleName, userDN)	425
setRoleById(roleName, userId)	425
setTitle(title)	426
WorkItem	426
assignTo(assigneeDnArray)	427
expire()	427
extend(newDate)	428
getAssigneesDN()	428
getCreationDate()	429
getCurrentActivityCN()	429
getExpirationDate()	429
hasExpired()	429
isStateActive()	430
isStateRunning()	430
isStateSuspended()	431
moveTo(activityName)	431
resume()	431
setExpirationDate(expDate)	432
suspend()	432
ContentStore	433
copy(srcURL, dstURL)	434
download(url, file)	434
exists(url)	435
getBaseURL()	435
getBaseURL(path)	435
getBaseURL(path, instanceid)	436
getContent (url)	436
getException(result_string)	437
getRootURL ()	438
getSize(url)	438
getStatus(result)	438
getVersion()	439

initialize(url)	439
isException(exception)	440
list(url)	440
mkdir (URLString)	441
move (String URLString1, String URLString2)	441
remove(url)	442
rmdir(url)	442
store (content, url)	443
upload(file, url)	443
CorporateDirectory	444
addUser (userDN, attributes, objectClasses)	444
deleteUserByCN(userCN)	445
deleteUserByDN(userDN)	446
deleteUserById(uid)	446
getUserByCN(userCN)	447
getUserByDN (userDN)	447
getUserById (uid)	448
modifyUserByCN (userCN, attrName, attrValue, operation)	448
modifyUserByDN (userDN, attrName, attrValue, operation)	449
modifyUserById (userID)	450
User	450
getId()	451
getDN()	451
Logging and Error Handling Global Functions	452
logErrorMsg (label, context)	452
logHistoryMsg (label, comment)	453
logInfoMsg(label, context)	453
logSecurityMsg (label, context)	454
setErrorMsg (errMessage)	455
Assignment, Completion, and Email Scripts	456
checkParallelApproval (dataField, stopAction)	456
defaultNotificationHeader()	457
defaultNotificationSubject()	457

emailByDN(DN)	458
emailById(userId)	458
emailOfAssignees()	459
emailOfCreator()	459
emailOfRole(roleName)	459
randomToGroup(groupName)	460
toCreator()	460
toGroup(groupName)	461
toManagerOf (userId)	461
toManagerOfCreator()	462
toManagerOfRole(role)	462
toParallelApproval(arrayOfUserDNs, dataField)	462
toUserById(userId)	463
toUserFromField(dataField)	464
Miscellaneous Global Functions	464
checkUserDNs(arrayOfUserDNs)	464
ejbLookup(jndiName)	465
evaluateTemplate(templateName)	465
expireIn(val, unit)	466
getAction()	467
getApplicationName()	468
getApplicationPath()	468
getApplicationPrettyName()	468
getBaseForFileName (processId)	468
getConnector(connectorKey)	469
getContentStore ()	469
getContentStore(httpURL,user,password)	470
getCreatorUserId ()	470
getCorporateDirectory ()	471
__getIncludePath ()	471
getJndiNamingContext()	471
getProcessInstance ()	472
getSubProcessInstance ()	472

getWorkItem ()	472
__includeFile (fileName)	473
mapTo(fieldName)	474
mount(jndiName)	474
setConnector(connKey, connObject)	475
__setIncludePath (includePath)	476
setRedirectionURL(stringURL)	476
url_OnDisplayHistory()	477
url_OnDisplayProcessInstance()	477
url_OnDisplayWorklist()	478
url_OnListApplications()	478
url_OnListEntryNodes()	478
Alphabetical Summary of JavaScript Objects	479
Appendix B Migrating from Previous Releases	483
Getting Started	483
Importing an Application to Process Builder	484
Assigning Exception Nodes	484
Checking for Errors	485
Deploying the Application	486
Migrating SSJS-specific Objects	486
Migrating Custom Fields	487
Appendix C Reserved Words	489
Glossary	491
Index	497

This *Developer's Guide* describes how to use Process Builder, a component of Netscape Application Server: Process Automation Edition (PAE). Read this guide to learn how to develop applications for PAE.

About This Guide

The information in this guide is organized into 18 chapters, 3 appendixes, a glossary. The first 10 chapters describe the fundamentals of using Process Builder:

- Chapter 1, "Introduction to Process Builder"
- Chapter 2, "Planning an Application"
- Chapter 3, "Creating an Application"
- Chapter 4, "Designing a Process Map"
- Chapter 5, "Defining Groups and Roles"
- Chapter 6, "Defining Data Fields"
- Chapter 7, "Designing Forms"
- Chapter 8, "Using Scripts"
- Chapter 9, "Setting Up Searching"
- Chapter 10, "Deploying an Application"

The next four chapters describe some of the sample applications included with PAE:

- Chapter 11, "The Data Sheet Application"
- Chapter 12, "The Office Setup Application"
- Chapter 13, "The Loan Management and Credit History Applications"
- Chapter 14, "The Insurance Claim Processing Application"

The next four chapters describe advanced topics. To get the most out of these chapters, you should be familiar with JavaScript and Java programming.

- Chapter 15, "Advanced Techniques for Scripting"

- Chapter 16, “Scripting with EJB Components”
- Chapter 17, “Writing Custom Activities”
- Chapter 18, “Writing Custom Fields”

The remaining parts of this guide are reference material:

- Appendix A, “JavaScript API Reference”
- Appendix B, “Migrating from Previous Releases”
- Appendix C, “Reserved Words”
- Glossary

Assumptions

This guide assumes you have installed PAE on your system. For installation instructions, see the *Installation Guide*.

It is recommended that you understand JavaScript. In addition, for some advanced topics, such as writing custom activities or custom fields, you must know how to program in Java.

Conventions Used in This Guide

File and directory paths are given in Windows format (with backslashes separating directory names). For Unix versions, the directory paths are the same, except slashes are used instead of backslashes to separate directories.

This guide uses URLs of the form:

`http://server.domain/path/file.html`

In these URLs, *server* is the name of server on which you run your application; *domain* is your Internet domain name; *path* is the directory structure on the server; and *file* is an individual filename. Italic items in URLs are placeholders.

This guide uses the following font conventions:

- The monospace font is used for sample code and code listings, API and language elements (such as function names and class names), file names, path names, directory names, and HTML tags.

- *Italic* type is used for book titles, emphasis, variables and placeholders, and words used in the literal sense.
- **Boldface** type is used for glossary terms

Viewing Documentation Online

For your convenience, PAE manuals are replicated online in both PDF and HTML formats. You can access the online documentation from the Help menu of each PAE component. You can access context-sensitive documentation by clicking a Help button or link in each PAE component.

Introduction to Process Builder

This chapter provides an overview of Process Builder, along with basic instructions for starting and using it.

This chapter contains the following sections:

- About Processes and Process Builder
- About Applications
- Starting Process Builder
- Using Process Builder

About Processes and Process Builder

This section contains the following topics:

- What is a Process?
- What is Process Builder?
- Who Should Use Process Builder?
- What is a Process Instance?

What is a Process?

A process is a series of steps that can be completed by one or more end users who participate in the process. For example, your company may have a process for requesting vacation time. The process participants would be an employee who makes the request, the manager who must approve it, and the payroll department who updates the employee's records.

What is Process Builder?

Process Builder is a component of Process Automation Edition (PAE), a solution for creating and deploying business process applications. For information about PAE components and their relationships, see the *Administrator's Guide*.

Process Builder is the graphical tool with which you create, maintain, and deploy web-based applications. These applications can handle forms-based processes and can also integrate with external systems.

Who Should Use Process Builder?

Process Builder is for people who design and deploy PAE applications. We refer to these people as process designers. As a process designer, you'll need to know JavaScript. In addition, if you want to extend PAE applications so that they can access external systems, you'll need to program in Java.

What is a Process Instance?

You create a separate application for each process. For example, submitting a vacation time-off request and submitting an expense report are two different applications.

When you deploy applications to a cluster, end users can access them. For more information about clusters, see the *Administrator's Guide*.

When users submit their first request in a PAE application, they initiate what is called a process instance. Each user's request generates a unique process instance. The point where a user initiates a process instance is called an entry point. The process instance ends at an exit point.

About Applications

This section contains the following topics:

- Sample Applications
- Creating an Application
- Deploying an Application

Sample Applications

Process Builder includes a set of sample applications that you can use as models when you create your own applications. For information on using and configuring them, see the following chapters:

- Chapter 11, “The Data Sheet Application”
- Chapter 12, “The Office Setup Application”
- Chapter 13, “The Loan Management and Credit History Applications”
- Chapter 14, “The Insurance Claim Processing Application”

Creating an Application

To create an application using Process Builder, you create a graphical view of the application's steps, from beginning to end. This view is called the process map. You must also define everything that relates to the steps, such as their forms, data fields, scripts, and access to forms.

In summary, you must create the following application elements:

- the process map, which defines the steps in the process

- the data dictionary, which captures process data
- the HTML forms, along with the people who can access these forms
- the scripts that perform functions within an application
- the groups and roles of the people who will be using the application
- the content store, which specifies where files are stored on the Enterprise Server

Application elements are represented in a window called the application tree view. This window resembles a typical file system view: it uses folders to represent element categories, and the folders contain the corresponding application elements. Initially, the folders in the application tree view are empty, or they contain default values.

To create the steps in the process, you drag icons from the Palette onto a process map. You connect steps by drawing arrows between them. These connections are called transitions.

As you construct the process map and insert new data fields, forms, and scripts, these items are also added to the application tree view. In this way, the folders are filled with items represented on the process map. Since all application elements are represented in the application tree view, you can easily access and edit any element's properties.

Deploying an Application

Once you have created an application, you can save it to a local folder, or deploy it to a cluster.

When you save the application on your local machine, all your definitions (the process map items, the data dictionary, the forms, etc.) are stored in a local applications folder in the `builder` folder.

Deployed applications are stored in clusters. When you launch Process Builder, you get a list of all the clusters specified in the `preferences.ini` file, as well as a list of any applications saved locally in the default directory.

When you deploy the application, you copy all its definitions to the cluster as entries in the configuration directory. There are two deployment stages: development and production. Deploy to development if you are still testing the

application and would like to be able to make the fullest possible range of changes. Deploy to production when you are ready to give end users access to the application.

For information on deploying applications, see Chapter 10, “Deploying an Application.”

Starting Process Builder

Before you can use Process Builder, you must install it on your local system. For installation instructions, see the *Installation Guide*.

The preferences.ini File

The `preferences.ini` file specifies the available clusters and the corporate user directory, as well as some of the folders that appear by default in your initial Process Builder window. The installer puts `preferences.ini` in the `builder` folder.

After the administrator creates a cluster, you need to add lines to the file specifying the cluster(s) and the corporate user directory you are accessing. Process Administrator displays these lines when a cluster is created, so you can cut them from there and paste them in your `preferences.ini` file. Otherwise, use the examples below to edit your `preferences.ini` file by hand.

Specifying a cluster

To access a cluster from Process Builder, add a line to the `preferences.ini` file in the following format. (Although shown below as multiple lines, be sure to type it as one line in the file, with no space after the `@` symbol.)

```
cluster = ldap://Unrestricted_User:Password@
Configuration_Directory:port/clusterDN
```

For example:

```
cluster = ldap://cn=Directory Manager:password@
directory.mozilla.com:389/cn=cluster, o=airius.com
```

Specifying a corporate directory

To access a corporate user directory from Process Builder, add a line to the `preferences.ini` file in this format:

```
corp_dir = ldap://corporate_Directory_Server:port/corp_Directory_Suffix
```

For example:

```
corp_dir = ldap://directory.mozilla.com:389/o=airius.com
```

You may want to add a `corp_dir` line to your `preferences.ini` file even if you do not know what cluster you will be deploying to. By adding a `corp_dir` line, the corporate user directory is available to you as you design applications, even without a deployment cluster.

Typically, your corporate directory does not require user authentication. However, if your corporate directory requires authentication, you can add a line in this format (be sure it's all on one line):

```
corp_dir = ldap://Unrestricted_User:Password@
corporate_Directory_Server:port/corp_Directory_Suffix
```

Warning The `preferences.ini` file also includes parameters that define the Java Virtual Machine. Do not modify this part of the file.

Adding or Changing Local Application Folders

You can also change some of the default local application folders in `preferences.ini`. The file ships with the following lines defining folders:

```
local_folder = Sample Applications; Samples
local_folder = Scratch Folder; Tmp
```

These two lines define two folders in your builder folder. The Samples folder stores the sample applications that ship with the product, and the scratch folder is a folder you can use for temporary content.

To add additional folders, use the following syntax:

```
local_folder = name_of_folder; pathname_of_folder
```

The pathname assumes you start in the builder folder.

Starting Process Builder on Windows NT

When you install Process Builder on your Windows NT machine, you choose a folder to install it in. By default, this folder is `NAS_Root/builder`, `NAS_Root` indicates the root directory of your Netscape Application Server installation.

To start Process Builder, go to the folder where you installed Process Builder and double-click on the `PMBuilder.exe` file. This file launches Process Builder.

Starting Process Builder on Solaris

When you install Process Builder on your Solaris machine, you choose a folder to install it in. By default, this folder is `NAS_Root/builder`, where `NAS_Root` indicates the root directory of your Netscape Application Server installation.

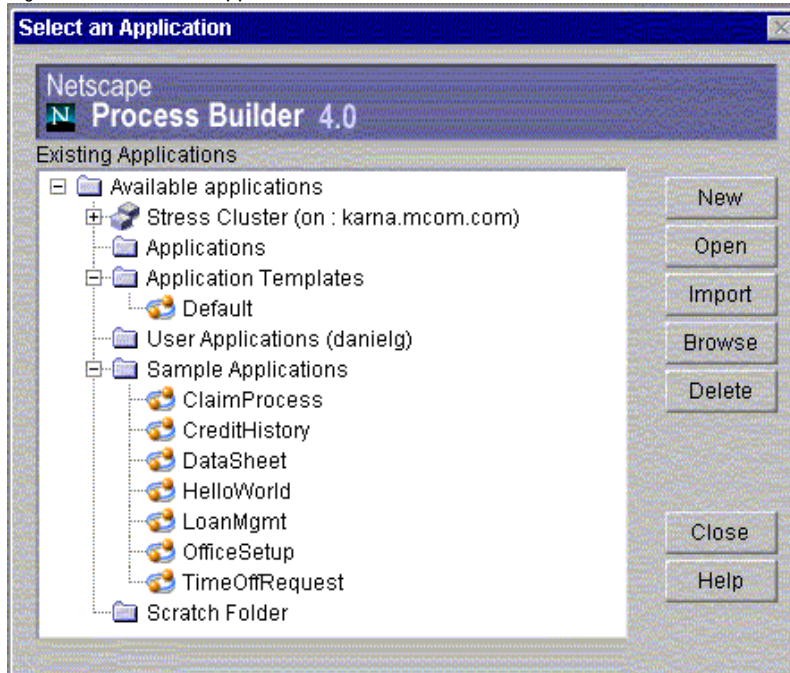
To start Process Builder, go to the directory where you installed it and type the following at the command prompt:

```
PMBuilder.sh
```

Using Process Builder

When you start Process Builder, you see a window like the one shown in Figure 1.1.

Figure 1.1 Select an Application window



In the left pane, this window lists all clusters, the applications deployed to them, and applications saved locally in the application directories. The applications in the clusters have labels indicating whether they were deployed in the development stage or in the production stage.

Before a cluster appears in this window, the PAE administrator must create it, and you must have edited your `preferences.ini` file to contain the correct cluster information. For more information on clusters in your environment, contact your administrator. Some of the local folders are also defined in the `preferences.ini` file.

For more information about the `preferences.ini` file, see “The `preferences.ini` File” on page 31.

The following local folders are provided by default:

Applications The main folder for storing applications.

Application Templates The folder where you store applications that you base other applications on. For example, if your applications share many elements in common, you might want to designate a template application, save it to this folder, and use it as a basis for future applications. A sample template application, `Default`, is included in this folder as an example.

User Applications A folder for applications stored by user. On Unix, this folder is your default user home directory.

Sample Applications A folder that contains all the applications shipped as samples with Process Builder. This folder is defined in your `preferences.ini` file.

Scratch Folder A folder for storing temporary, or scratch, versions of your applications. This folder is defined in your `preferences.ini` file.

The right pane contains the following buttons:

New Creates a new application.

Open Opens the application selected in the tree.

Import. Use this button to import an existing application into your Process Builder. The application must be a `.zip` file, for example, an application zipped and sent via email. The import function unzips the application into the selected folder and opens the application for editing.

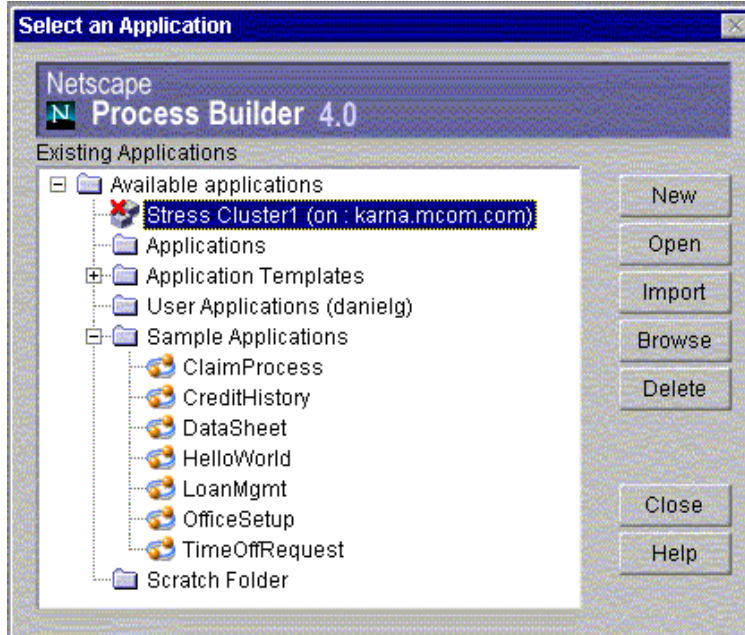
Browse Navigates through the file system to open applications stored outside your local applications folders. Choose the LDIF file of the application you want to open. (LDIF is a file format for storing directory entries). LDIF files have the same names as the application and the folder the application is stored in.

Delete Deletes the selected application. You cannot delete applications that have been deployed. Deployed applications can be deleted only by the administrator.

Cancel Closes the window, but leaves Process Builder running.

Sometimes a cluster icon has a red “x” in it, as shown in Figure 1.2:

Figure 1.2 Cluster error



This icon indicates that Process Builder is unable to access a cluster, and appears when Process Builder cannot access the configuration directory. If you see this icon:

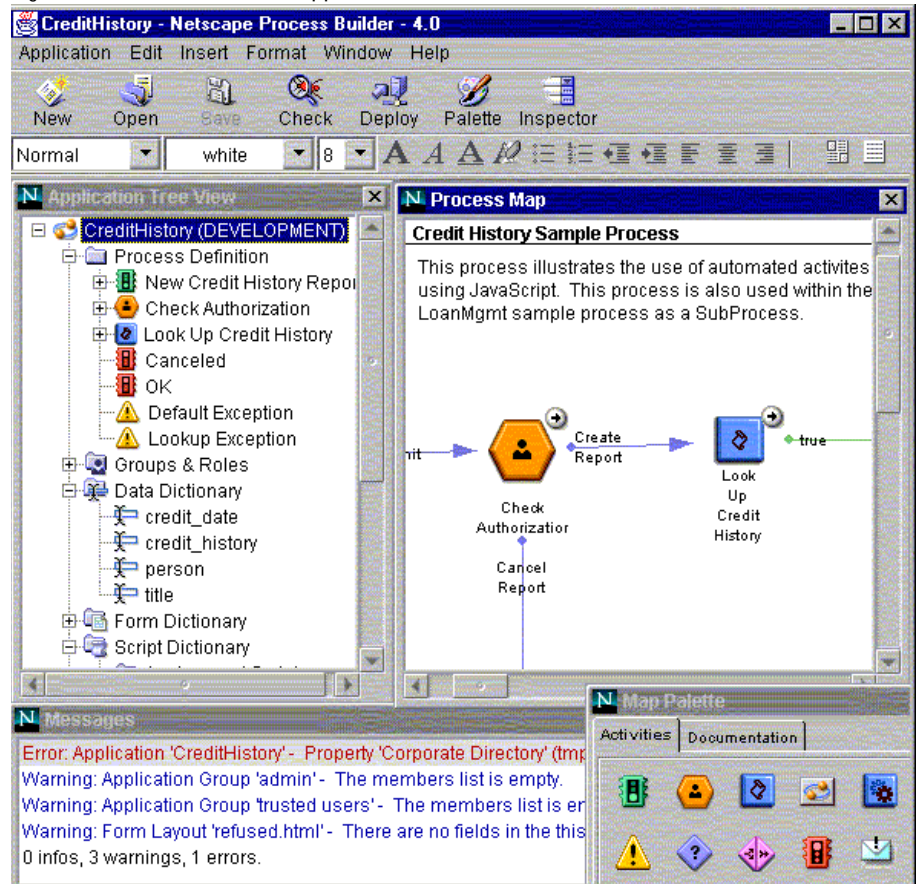
- verify that the cluster information in `preferences.ini` is correct.
- verify that the configuration directory is up and running

Your administrator can also help you solve cluster problems.

Process Builder Applications

Figure 1.3 shows a sample view of Process Builder when you open an existing application (the `CreditHistory` application in this case).

Figure 1.3 A Process Builder application



Within each application, Process Builder provides the following views and tools:

- tree view of your application's elements
- process map
- a map palette
- Messages window
- main toolbar
- menu commands

- HTML editor toolbar

You use these views and tools to create and edit applications. In addition, some commands are available by right-clicking on an element in the application tree view or process map.

Application Tree View

The application tree view lists all the elements in an application. Each application has a set of default elements, shown as a set of standard folders and icons. When you create a new application, the element folders are empty or contain defaults. As you construct the process map and insert new data fields, forms, and scripts, the folders fill up with subitems that you can directly access at any time to inspect or customize their properties.

These are the default folders and icons:

Process Definition This folder contains all the items in an application. See “Map Palette” on page 39 for more information on these items.

Groups and Roles This folder contains the application’s groups and user roles. See Chapter 5, “Defining Groups and Roles,” for more information.

Data Dictionary This folder contains the data fields created for the application. These fields define data that is to be captured when users run the application. See Chapter 6, “Defining Data Fields,” for more information.

Form Dictionary This folder contains the HTML forms created for the application. Forms define how information is presented to a user. See Chapter 7, “Designing Forms,” for more information.

Script Dictionary This folder contains the JavaScript scripts available to the application, organized by script type. Scripts are either predefined or created by you. See Chapter 8, “Using Scripts,” for more information.

Form Access This icon brings up a window which lets you specify which HTML form is displayed for a specific role during each step. You set up access to forms by dragging forms from the application tree view into the window. See Chapter 7, “Designing Forms,” for more information.

Content Store The content store icon brings up a window where you specify the URL on the Enterprise Server where you store attached files, and also the public user that can access these files. When end users attach files to forms as they complete steps in the process, the files are stored in the content store. See Chapter 6, “Defining Data Fields,” for more information.

Process Map

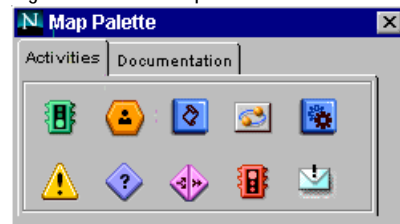
The process map provides a visual representation of an application. It shows the steps needed to complete the process, and how they are connected to each other.

When you create a new application, the Process Map window is blank. If you are revising an existing process, the current version of the process appears in the Process Map window. To design a process in the Process Map, you drag activity icons from the palette and connect them with transitions.

Map Palette

Figure 1.4 shows the Map Palette, also called the palette. When you create a new application or when you open an existing one, the palette is open by default. You can also click the Palette button to open it.

Figure 1.4 The Map Palette



By default, the palette has two tabs: Activities and Documentation. If you use custom activities, you may also have one or more custom tabs on your palette. For more information on adding custom palettes, see “Adding a Custom Palette” on page 95.

Activities Tab

The Activities tab displays icons that you can drag onto the process map to design a process application. Each of the palette items represents a type of step in the process, a notification, or an error response. Table 1.1 shows the icons on the palette and what they represent.

Table 1.1 Icons in the Activities tab











Icon	Description
	Entry Point. A point at which a user can initiate a process. An application can have several entry points. For example, if the first few steps create an ID number for the user, a returning user who already has an ID can skip those steps.
	User Activity. A step within the process that requires a user to perform a task. Each user activity has an assigned user who performs the task (assignee) and a form the user needs to fill out in Process Express. After you place activities in the process map, you define the sequence in which they are to be executed by connecting them with transition lines.
	Automated Activity. An automated step performed through a JavaScript script without user intervention.
	Subprocess. A step that calls a process from within another process. The process that calls the subprocess is considered to be the main, or parent process, and the subprocess is considered to be its subordinate or child process. A parent process can have several children processes, each of which is a stand-alone process complete with entry and exit points.
	Exception Manager. A step that allows the administrator to intervene manually if errors occur when users run the application.
	Decision Point. A conditional step that causes the process to use different steps based on a condition. For example, you might have a decision point that directs the process to different steps depending upon the cost of an item.
	Split-Join (parallel processing). A step within the process that branches in two or more branches so that two or more activities can execute in parallel.
	Exit Point. A step at which the process ends. An application can have several exit points. For example, in a vacation time off request application, an exit point could be the approved vacation request, and another could be a vacation that was not approved.

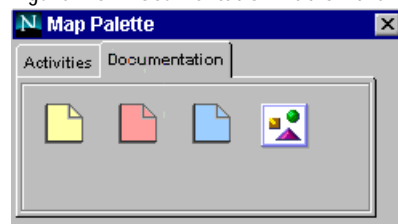
Table 1.1 Icons in the Activities tab

Icon	Description
	Custom Activity. A step at which a PAE application connects to external components or services.
	Notification. An email notification that is triggered when a user activity is started. The email can serve many purposes. For example, it can inform the person who started the process or other users of the process's progress.

Documentation Tab

Use this tab (shown in Figure 1.5) to drag documentation notes or graphics to the process map.

Figure 1.5 Documentation Tab on the Map Palette



For example, different developers may include notes on a process map coded by color, or the notes could explain features of the process for use when you roll-out the process.

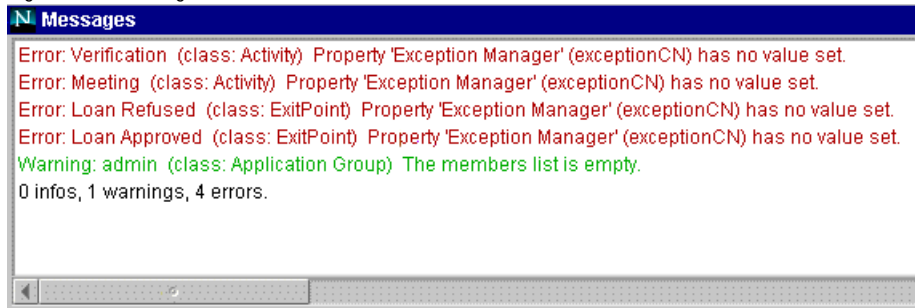
You can also add images to a process map by dragging the image icon to the map and then browsing to the image you want to include.

Messages Window

The Messages window is where error, warning, and information messages are displayed when you check the application syntax using the Check button. The syntax is also checked automatically when you attempt to deploy an application.

The window is blank until the syntax is checked, after that it contains messages, as shown in Figure 1.6.

Figure 1.6 Messages Window



You must fix all errors before you can deploy an application. You can deploy an application that contains warnings, but you may encounter difficulties later when you try to use the application.

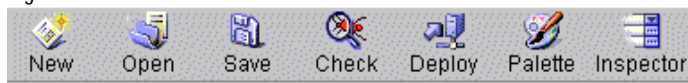
The last line of the window summarizes the information found by the syntax check.

If you right-click in the Messages window, a dialog box appears. You can use this dialog box to selectively display different types of messages (for example, errors only or warnings only). You can also clear the Messages window.

Main Toolbar

The main toolbar (see Figure 1.7) contains frequently-used commands.

Figure 1.7 Process Builder main toolbar



The toolbar contains the following buttons:

New Launches the New Application dialog box, where you create a new application which is not based on a previous application.

Open Opens an existing application.

Save Saves the application locally. Use this command to save your work as you are creating or editing an application.

Check Checks the application code and displays errors, warnings, and information messages in a separate window. You need to fix all errors in your application before you can deploy it. You can deploy an application for which you receive warnings, but it may lead to problems in the future. The last line of the window gives you a summary of how many of each type of message the check produced.

Deploy Deploys the application to a cluster.

Palette Displays the palette.

Inspector Displays the Inspector window for the selected element. This window allows you to set the properties of any element. You can also double-click the element's name in the application tree view to open the Inspector window. Some elements have evaluation order Inspector windows as well as property Inspector windows. Typically, an evaluation order window allows you to reorder conditions or roles into a specific hierarchy or sequence.

Menu Commands

Several menus provide commands for use as you design a PAE application. The menus are Application, Edit, Insert, Format, Window, and Help.

Application

New Creates a new application. If you have an application open, this closes the current application after asking if you want to save it first, and launches a new application with a fresh tree view and an empty process map.

Open Takes you to the “Select an Application” dialog box, from which you can open an existing application. If an application is already open, Process Builder closes the application. If the open application has unsaved changes, you are first prompted to save the application.

Import from ZIP Imports an existing zipped application. If you have an application open, this closes the current application after asking if you want to save it first. Browse to the zip file and import it into the current Application folder.

Delete Takes you to the “Select an Application” dialog box, from which you can delete an application. If an application is already open, Process Builder closes the application. If the open application has unsaved changes, you are first prompted to save the application.

Save Saves the currently displayed application to your local machine.

Save As Saves the current application with a different name.

Import File Brings up a dialog box, from which you can copy a file into the application’s directory. Enter the path name of the file to import, or use the Browse button to browse to it. Then specify where you want the file copied. The file is automatically copied to a specific location, depending on the file category you select. (See Table 1.2.) If a subdirectory doesn’t exist, it will be created.

Table 1.2 Destination folders for imported files

File Category	Destination Folder
Top Folder	The same directory as the current application. For example: <code>builder/Applications/app_name</code>
Client Images	<code>images</code> subdirectory
Client JavaScript	<code>js</code> subdirectory
Server JavaScript	<code>WFScripts</code> subdirectory
Server JAR Files	<code>lib</code> subdirectory
Mail Notification Templates	<code>templates</code> subdirectory

Check Errors Checks the application code and displays errors, warnings, and information messages in a separate window. You need to fix all errors in your application before you can deploy it. You can deploy an application for which you receive warnings, but it may lead to problems in the future. The last line of the window gives you a summary of how many of each type of message the check produced.

Deploy Deploys the application to a cluster.

Save Process Map as JPEG Saves the process map as a .jpg file. Use this option when you want to show the whole process map outside of Process Builder. For example, by saving the process map as an image, you can print it or insert it into an HTML document. This option saves the image as *application_name.jpg* in the folder of the application.

Exit Quits Process Builder. If you have unsaved changes in an application, it gives you the option of saving them.

Edit

Cut Cuts an HTML layout element from a form, or cuts text from a script.

Copy Copies an HTML layout element into a form, or copies text into a script.

Paste Pastes an HTML layout element into a form, or pastes text into a script.

Delete Deletes a single element such as an activity, a data field, or a form from the application tree view. You cannot delete folders from the application tree view, or default item such as predefined scripts and default users and groups. Also, if the application has been deployed in the production stage, you may not be able to delete certain items.

Insert

Group & Role Inserts a new group or role.

Data Field Inserts a new field.

Form Inserts a new HTML form.

Script Inserts a new script.

HTML Element Inserts the HTML element that you select from the associated submenu. For more information, see “Insert HTML Element” on page 167.

Format

These are standard HTML editing commands. Use them to edit HTML forms in the HTML editor.

Size Applies the font size you select to the text.

Style Formats text as bold, italic, underline, or strikethrough.

Remove All Styles Removes size and style formatting from text.

Heading Applies an HTML heading tag to the text.

List Applies an HTML list tag to the text. You can choose either a bulleted list or a numbered list.

Align Aligns text at the left, right, or center of the page.

Decrease Indent Decreases the amount of indenting on the text.

Increase Indent Increases the amount of indenting on the text.

Window

Select one of these menu items to display the window, or, if it is already displayed, bring it to the foreground.

Project Window Displays the application tree view.

Process Map Displays the process map.

Palette Window Displays the palette.

Messages Window Displays the error and informational messages produced when you check the application syntax using the Check button.

In addition, you can reorganize the windows with either of the following menu items:

Auto Arrange Windows Arranges windows to fit side-by-side within the main window. Four windows will be displayed: the application tree view, process map, Messages window, and the palette. The resulting arrangement is not updated when you resize the main Process Builder window.

Always Arrange Windows Similar to Auto Arrange Windows, except that the arrangement is always updated. Whenever you resize the main window, the individual component windows are resized to maintain their side-by-side arrangement.

Help

Help contents Accesses Process Builder full table of contents. Click a link to go to the topic you are interested in, or use the Index to find a topic. If you click the Contents button within the window, you can access the “Other Documentation” link, which gives you access to the PAE documentation set.

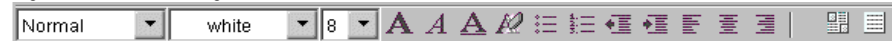
Custom Help Displays help you have customized for Process Builder at your site. For example, you might use this feature to add links to your Process Express site or to explain site-specific customizations to Process Builder. To add your own help, edit the file `custom_help.html`, found in the `support/html` directory of your PAE installation.

About Displays the software version information.

HTML Page Editor Toolbar

The HTML Page Editor toolbar allows you to format the HTML forms that you create. Figure 1.8 shows the HTML Page Editor toolbar.

Figure 1.8 HTML Page Editor toolbar



The drop-down lists and the icons assign HTML formatting to the text you select in your HTML forms. See “Using the HTML Page Editor Toolbar” on page 166 for more information.

Planning an Application

This chapter describes the issues to consider when you are planning an application.

This chapter includes these sections:

- Planning Overview
- Planning the Process Map
- Planning Assignments
- Determining the Data Requirements
- Planning Forms
- Planning Custom Scripts
- Planning Searches
- Planning Deployment

Planning Overview

Process Builder provides a powerful and intuitive visual environment to create and deploy PAE applications. This chapter describes the things to think about before you develop your application. For an explanation of completed applications, see the sample application chapters in this book.

To plan your PAE application, you need to do the following tasks:

1. Decide on the sequence of steps involved in a process and whether you need to create subprocesses or custom activities.
2. Define who the people involved in the process are and what roles they play.
3. Decide whether you want to give people who aren't directly involved in the process access to some of the process information.
4. Decide what data you want the application to track on forms, internally within scripts, and what you need to pass to subprocesses and to other programs using custom activities.
5. Define a useful set of forms to allow users to complete each step, and if you like, to allow users to monitor the process.
6. Decide which users and groups should have access to which forms.
7. Decide if you need to write any special scripts for your application.
8. Decide if you want users to be able to search the application, and if so, for which information.

Planning the Process Map

Before you can create a PAE application, you must understand the process you want the application to handle.

The best way to create an application is to examine your current process, automated or manual, figure out the steps involved, and then improve the process with PAE's automation and database capabilities.

For more information on Process Maps, see Chapter 4, “Designing a Process Map.”

Entry Points

After analyzing the existing process, you should decide at what point(s) in the process you want to start your application. You can have more than one entry point, depending upon the process, but you need to find starting points that still let you capture all the data you need to capture. If you start halfway through a process, you may not have all the data you need to carry the process through.

For example, if you have a process for posting documents on a web site, you might have one entry point for documents that need to be converted to HTML, and another entry point farther through the process for documents that are already in HTML.

For more information on entry points, see “Entry Points” on page 77.

User Activities

Each step in the process that a user needs to perform is called a user activity. When you figure out the steps in the process, you need to decide which steps need to be performed by a person and which can be performed automatically.

For example, in the application that handles an employee’s vacation request, you might have three activities:

- Manager approval, where the manager either approves the request or asks for clarification.
- Clarification, where the employee provides any requested clarification
- HR approval, where HR approves the request

Since these all require a user to take action, they are all user activities, not automated activities. For more information, see “User Activities” on page 78.

Automated Activities

If a step can happen automatically, you'll want to create an automated activity, instead of an activity that requires a user to perform an action.

For example, the Data Sheet sample application uses an automated activity to build the data sheet. The automated activity runs a script that looks up price information and creates an HTML form based on this information. This saves a user from having to enter the data. The next step of the process is an activity that presents a user with the data for approval.

For more information on automated activities, see "Automated Activities" on page 84.

Custom Activities

You need to determine if you want to create a custom activity to connect to external services, to integrate information from other services into your PAE application, or to extend Process Manger to other services. For example, you can use custom activities to connect to Netscape Application Server Extensions for CICS or SAP. A developer needs to write a Java class, write an XML file that describes the Java class, and put the two together in a .zip or .jar file. You need to decide what data you need to transfer to the external component, and what data you need to receive back.

For more information on custom activities, see "Custom Activities" on page 91.

Subprocesses

You need to determine if you want to create a subprocess that can be used across your environment. A subprocess is a fully functional PAE application that can be called from within another PAE application. The Loan Management sample application, which controls the process of approving a loan, calls the subprocess Credit History. Credit History can be run either as a subprocess of the Loan Management application or as a stand-alone application. For more information on subprocesses, see "Subprocesses" on page 86.

Decision Points

You need to determine if the process has a single set of steps, or if it branches because of conditions. Decision points are where the process branches. For example, in the Data Sheet sample application, there is a decision point where if the price of the product on the data sheet is less than \$1000, the data sheet is immediately published. However, if the price is greater than or equal to \$1000, the data sheet must be approved by the Vice President before being published.

For more information about decision points, see “Decision Points” on page 100.

Parallel Processing

You need to determine if you want to use parallel processing, which allows your process to have two or more branches so that two or more activities can execute in parallel. In the Office Setup sample application, which controls the process of setting up an office for a new employee, each subtask is grouped into a processing branch that progresses independently of the other subtasks. For example, the MIS department can set up the phone while the purchasing department is ordering the computer. Problems completing one task won't affect the progress of a parallel task. In complex processes, there can be several levels of nested parallel branches.

For more information about parallel processing, see “Split-Join (Parallel Processing)” on page 101.

Exit Points

You also need to decide where to end the process. You may have more than one exit point. For example, in the Time Off Request sample application, an employee's request for vacation time can end three ways: the vacation is approved or denied by management, or cancelled by the employee. Those three outcomes translate into three exit points in the application.

For more information on exit points, see “Exit Points” on page 105.

Transitions

Once you have the basic steps of the process in place, you need to determine the flow between them. The connections between steps are called transitions, and are represented on the process map by arrows connecting the steps. Since a decision point requires the process to branch, each branch has a transition leading to a different step.

For more information on transitions, see “Transitions” on page 106.

Notifications

Once you have the steps and transitions in place, you need to think about where in the process you want to set up email notifications, what information the notification contains, and who needs to receive the notifications. The notifications are sent as soon as the process reaches the step.

For example, in the Time Off Request sample application, the employee is notified if the vacation is approved or denied at the exit points for approved vacation and denied vacation. At the exit point where the employee cancels the vacation, no notification is required.

For more information on notifications, see “Notifications” on page 104.

Exceptions

Exception handling allows the administrator to intervene manually if errors occur. Every process step (other than an entry point) must be assigned an Exception. See “Exception Manager” on page 97 for more information on exception handling.

Planning Assignments

You need to look at each activity and determine who needs to do it. The person who performs an activity is called the assignee. You determine an activity’s assignee by asking the following questions:

- Who needs to perform this step?
- Should this step be assigned to a single individual, or to a group of individuals so that any one of the group can perform the work item?
- Is this an activity where a number of individuals need to approve some information before the process continues (parallel approval)?
- Does the person who performs the activity vary depending upon the process instance? For example, a vacation request might have an activity for the manager's approval. The manager who needs to approve the vacation, depends on which employee requested the vacation for a particular process instance. In other cases, the assignee might be the same regardless of the process instance. For example, if every purchase over a certain amount needs to be approved by the Chief Financial Officer, the assignee of that approval activity is the same regardless of the process instance.
- If the assignee depends upon the particular process instance, what information do you need to have in order to determine who the assignee should be?

Once you have the answers to these questions, you know who needs to be assigned to activities, and that information helps you set up the groups and roles you need for an application.

For more information on assignments, see “Setting Activity Assignments” on page 81.

Planning Groups and Roles

Once you have figured out who needs to participate in the process and what they need to do in the process, you need to think about your groups and roles. If a user activity can be performed by any one person in a group, you must set up a group to assign the activity to.

For example, in the Office Setup sample application, one user activity is “Order Computer.” Anybody in the purchasing group can perform this step, so the activity is assigned to a group called Purchasing.

If you have an assignee that varies by process instance, and the information needed to identify this person is stored in a data field for each process instance, you can set up a field role. For example, in a process where a web designer is

working with a graphic artist on a web page, you could create a field role for the artist. The form initiating the web page art could have a field on it where the designer fills in the artist's name. The application then stores the artist for a particular process instance, and uses it throughout the process.

To figure out what groups and roles you need, ask the following questions:

- Who in the company can initiate a process?
- If approval is needed, what needs approval and who needs to do it?
- If you need to use a group of people in your application, is the group already defined in the corporate user directory?

There are four default groups and roles for an application:

- creator (the person who initiates a process instance)
- admin (people allowed to administer the application)
- all (any interested party)
- trusted users (people allowed to connect to the subprocess)

If you need additional groups and roles, you need to create them. If the group already exists in your corporate user directory, you can create a group for your application that is tied to the group in the corporate user directory.

For more information on groups and roles, see Chapter 5, “Defining Groups and Roles.”

Planning for Monitoring

In addition to the assignees in an application, you can also let other people who aren't directly involved in the application monitor the progress or view information on a read-only basis. If you want to give these nonparticipants access to information, you may need to create additional groups and roles.

Planning Delegations

For some activities, you may want the assignee to be able to delegate the work item to another user. If you want the assignee to be able to delegate, you must set the activity's Allow Delegation property to yes. Before allowing delegation, though, you need to think about whether the activity should be delegated. For example, if you had a process for approving departmental salary increases, you may want the head of the department to approve the increases personally. In this example, you would not allow delegation.

Determining the Data Requirements

After you have defined the steps in your process, you need to define what data you need to track. Once you've determined the data you need, then you can create the process' data dictionary using Process Builder. The following questions help you determine what data fields you need:

- What information do assignees need to see to perform each step?
- What information do you need to gather from assignees at each step?
- What data do you want to have a record of?
- What information do the scripts in the application require?
- If you're using subprocesses, what information is passed between a parent and child process?
- If you're using a custom activity, what information is passed between the PAE application and the external component or service?

For each data field you create, you need to think about how to present it to users on the form (for example, is it a text field, radio buttons, or a file users need to attach?). You also need to think about how it will be stored in the database (is the data type text, a date, or an integer?)

For more information on data fields, see Chapter 6, "Defining Data Fields."

If the standard data field classes provided with PAE do not fit your needs, you can create your own. For example, you can use custom data fields to access external data sources and to generate dynamic content at entry points. To use custom data fields you need to write Java classes that implement the field.

File Attachments

Some information your application gathers may be files users contribute by attaching them to forms. You create specific data fields for file attachments. If you are using file attachment data fields in your application, you must plan for their use:

- You must define a URL on your Enterprise Server to store these documents.
- You must set the Enterprise Server's access privileges for the URL, so that the documents are available to a public user.

Setting up the public user allows all users to upload documents to that area. Web Publishing must be turned on in your Enterprise Server.

For more information on file attachments and the Enterprise Server, see “Setting Up the Content Store” on page 156.

Planning Forms

Once you have thought through the process steps, the data requirements, and the people involved you need to think about the forms to use to collect the data, move it from person to person, and display the data to people involved in the process.

You can design a different form for each role or group at each step in the process. Different forms let you display only the information needed by a particular person or group. For example, the assignee of an activity often has a special form which has more information on it than the forms for other people involved in the process, or people who are monitoring the process. Often the creator of the process instance has forms so he or she can follow the process instance's progress.

You create forms in a form wizard in Process Builder. You add fields to the forms in a form editor by dragging them from the data dictionary to the form. You need to decide which fields to put on a form, and whether they can be edited by the assignee or not.

For more information on forms, see Chapter 7, “Designing Forms.”

Planning Access to Forms

Using form access you control which users see which forms. To plan form access for a process, use the following questions:

- At what step in the process is the form used?
- Who views the form (is the viewer the assignee, a participant in the process, or an observer?) If the viewer is a participant in the process, what group or role do they belong to?
- Do you want the form to be viewed by multiple groups and roles?

When you have the answers to these questions, you create the association between an activity, the role or group, and the form in the Form Access window.

For more information on form access, see “Setting Access to Forms” on page 172.

Planning Custom Scripts

Many of an application’s actions are performed by JavaScript scripts. For example, JavaScript assignment scripts determine which user is assigned to a step. Process Builder includes some standard scripts; however you may find you need to write additional JavaScript scripts to perform actions specific to your application. In planning these scripts, consider what kind of scripts you need to write and whether you already have similar scripts you can base them on. Scripts you plan to reuse or plan to call from other scripts are stored in the Toolkit folder.

For more information on using scripts, see Chapter 8, “Using Scripts.”

Planning Searches

If you want users to be able to search for information about process instances, or for specific field information, you need to design your application so that searching is allowed. Use the following questions to help you plan your searches:

- Who do I want to allow to search application data?
- What fields do I want to allow people to use for searching? For example, in an application for publishing documents to a web site, you might want to be able to search based on the author's name or the document title.

For more information on searching, see Chapter 9, "Setting Up Searching."

Planning Deployment

After you have completed the application, you may want to deploy it for testing before rolling it out to a production server. In test mode, the person who designed the application can do a walkthrough to see that all steps are correct and that the information flows through the system correctly. In test mode, all activities are assigned to the creator of a process instance, so you cannot test form access fully until you deploy in non-test mode. Only then does the application use all the groups and roles.

PAE has two deployment stages: development and production. When you deploy to development, you can still change everything about an application. However, once you deploy to production, some information about the application can no longer be changed.

After you have tested the application and made any necessary fixes, you can deploy it for use in real-world environments. Once the application is deployed to end users, it usually leaves the builder's control. However, even after end users are using it, the builder and the process administrator need to communicate whether the application needs changes, if it needs to be replaced, or if it is made obsolete.

For more information on deploying applications and editing deployed applications, see Chapter 10, "Deploying an Application."

Creating an Application

This chapter takes you through the process of creating a new application. First it outlines the process.

This chapter includes these sections:

- Application Creation Overview
- Getting Information from the Administrator
- Creating a New Application
- Using Sample Applications
- Deleting an Application

Application Creation Overview

After planning your process application, you're ready to create it. Here are the main steps:

1. Get information from the PAE administrator.

See "Getting Information from the Administrator" on page 63 for more information.
2. Launch Process Builder and create a new application.

See "Creating a New Application" on page 63 for more information.
3. Design a process map, including all activities that users need to perform.

See Chapter 4, "Designing a Process Map," for more information.
4. Define the groups and user roles for process steps participants and non-participants (observers).

See Chapter 5, "Defining Groups and Roles," for more information.
5. Define the assignees for a step.

See "Setting Activity Assignments" on page 81 for more information.
6. Define the data fields, and set up the file attachments you need.

See Chapter 6, "Defining Data Fields," for more information.
7. Design the forms to display the appropriate data fields and to attach the necessary files.

See Chapter 7, "Designing Forms," for more information.
8. Set the form access, thereby associating the form with the appropriate step and with the appropriate participants for that step.

See Chapter 7, "Designing Forms," for more information.
9. Decide which users, if any, are allowed to perform searches. Also decide which fields are searchable.

See Chapter 9, “Setting Up Searching,” for more information.

Getting Information from the Administrator

In order to successfully develop and deploy an application, you need to work with the PAE administrator to set certain values for the application. In general, information dealing with clusters, the Enterprise Server, and the database may be set by your administrator. For example you may need to work with your administrator to set appropriate values for the following fields:

- the application properties DB User and DB Password, the database user and password which has the privileges to create database tables
- the content store properties URL, Public User and Public Password. These properties are the URLs on the Enterprise Server to which the attached files are posted, and the user and password for accessing the URL
- the person or people allowed to administer the application (the people you add to the "admin" group)
- the PAE administration user and password needed when you deploy the application to a cluster

For more information on these values, see “Setting Application Properties” on page 65, “Setting Up the Content Store” on page 156, “Default Groups and Roles” on page 113, and “The Deploy Application Dialog Box” on page 206.

Creating a New Application

Since each process is handled by its own application, the first thing you need to do after planning the process is to create a new PAE application for it.

To create a new application, follow these steps:

1. Launch Process Builder.
2. In the Select an Application dialog box, click New Application.
3. Fill out the dialog box fields.

See “The New Application Dialog Box” on page 64 for details.

4. Click OK. This displays a default set of empty resource folders and a blank process map.

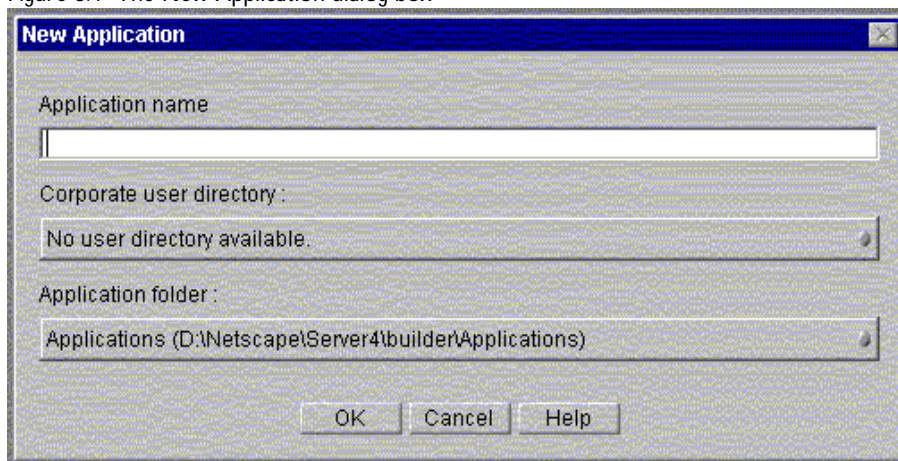
See Chapter 4, “Designing a Process Map” for details.

5. Set the application’s properties in the Inspector window.

The New Application Dialog Box

The New Application dialog box (see Figure 3.1) is where you enter the name and location of a new application. The location you enter here is the location on your local system. Typically you have a “working” copy of the application on your local system. Once you deploy an application, you generally use the deployed application as the basis of any modifications you want to make.

Figure 3.1 The New Application dialog box



Application name Enter a name for your application. The field is required. It can contain only the following characters: A to Z; a to z; 0 to 9 and _ (underscore). All other characters are not allowed.

Corporate user directory This directory contains all valid users and groups for the corporation, and is the source of the user information you need in order to set up user groups and roles for the application. The available directories that appear in the drop-down list are defined in the `preferences.ini` file.

Application folder This is the full physical path to where you want to save this application on your local machine. By default the path is the main applications folder on your local machine, `NAS_Root/builder/Applications`. The following folders are available from the drop-down list:

- Applications: The main folder for storing applications.
- Application Templates: The folder where you store applications that you base other applications on. For example, if your applications share many elements in common, you might want to designate a template application, save it to this folder, and use it as a basis for future applications. A sample template application, `Default`, is included in this folder as an example.
- User Applications: A folder for applications stored by user. On Unix, this folder is your default user home directory.
- Sample Applications: A folder that contains all the applications shipped as samples with Process Builder. This folder is defined in your `preferences.ini` file.
- Scratch Folder: A folder for storing temporary, or scratch, versions of your applications. This folder is defined in your `preferences.ini` file.

Setting Application Properties

The application has properties that you need to set. These properties affect the application as a whole.

To set the application properties:

1. Double-click the application's name at the top of the application tree view or select the application's name and click the Inspector icon, or right-click the application's name and choose Properties.
2. Fill in the fields.

See “The Application Properties Dialog Box” on page 66.

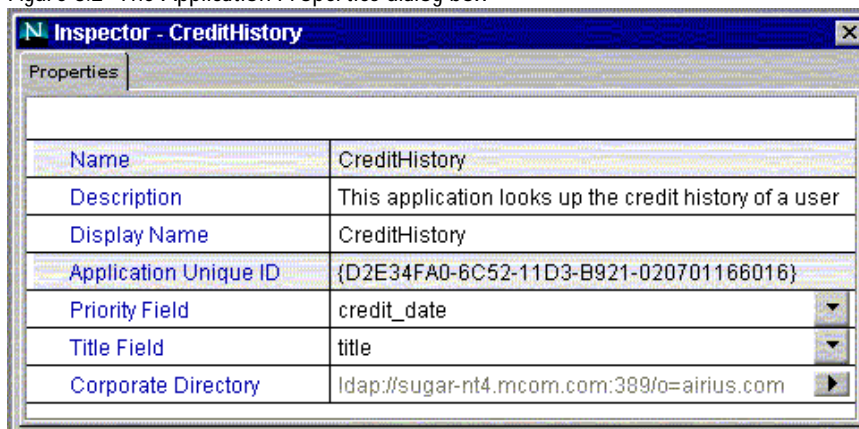
3. When you're done, close the window by clicking the close box.

The changes you make are saved automatically.

The Application Properties Dialog Box

The Application Properties dialog box (see Figure 3.2) includes the properties you set for the application as a whole. Note that you cannot change fields that are grayed out. The fields that are grayed out vary depending on whether you deployed your application in the Development or Production stage. For more information on deployment, see Chapter 10, “Deploying an Application.”

Figure 3.2 The Application Properties dialog box



Name The name of the application that appears in the list of applications. This field is required.

Description A longer description of the application. This description appears in Process Express. The user has to have enough information between the Display Name and the Description to identify the application they want from a list of applications, so it's important to make these two fields descriptive.

Display Name The longer, more descriptive name displayed by Process Express. If you are updating an existing application by saving it with a new application name, you can still keep the same Display Name, so that the name users see can be the same from version to version of the application.

Application Unique ID An internal unique ID generated automatically to track each application. This ID identifies the application, and differentiates applications with the same name. If you do a “Save As” on an application, you can save the application using the same name, but the unique ID will be

different. If you attempt to deploy an application that has the same name as an existing deployed application, but a different unique ID, you are prevented from deploying.

Priority Field To set up your application so that process instances can be prioritized, use this property. First you need to create a data field of class ID Select List to represent the priority. The field must have the value of an integer between 1 and 5 (1 is the highest priority; 5 is the lowest). Next you choose this field to be your priority field using this property. Then you put the priority field in a form so that the person responsible for prioritizing work items can select a priority value for the application's work items. If you do not use it, all process instances have a priority of 3 (normal). The behavior of your application is not affected by the value of this field.

Title Field The data field that contains the title of a process instance. This field is required. This field appears in Process Express as the title of the process instance, so you need to set this property to a field that has a value that describes the process instance and that end users will find helpful. This field will also be the title of any email notifications sent. It is often best to create a special field to be the title field. The field must already exist before you can fill it in here. This field is required.

Corporate Directory The corporate LDAP directory this application uses for user and group information. A pop-up window lets you choose a directory, either based on a cluster or by name. The listed directories are derived from the corporate user directory values in your `preferences.ini` file. The directory you choose must be the same as the directory of the cluster to which you will deploy the application. If you choose the directory based on a cluster, your directory will be up-to-date, even if the administrator changes the directory for the cluster. However, if the administrator changes the directory for the cluster, you have to make sure your users and groups correspond in both directories.

Using Sample Applications

PAE ships with sample applications that demonstrate how to build applications, as well as showing the kinds of tasks you can perform in an application. The available samples are listed in Table 3.1.

Table 3.1 Sample Applications

Application	Description
DataSheet	Creates a product data sheet, including setting pricing and getting signed approval by two levels of management. For more information, see Chapter 11, "The Data Sheet Application."
OfficeSetup	Sets up an office for a new employee, including assigning an office space, ordering a computer, and installing the phone and computer. For more information, see Chapter 12, "The Office Setup Application."
LoanMgmt	Submits an application for a loan, including launching a subprocess (CreditHistory) to check the applicant's credit history. For more information, see Chapter 13, "The Loan Management and Credit History Applications."
CreditHistory	Checks a loan applicant's previous credit history with the company. Used as a subprocess by the LoanMgmt application. For more information, see Chapter 13, "The Loan Management and Credit History Applications."
ClaimProcess	Submits an insurance claim, including checking policy details from a flat file, and going through the approval process at the insurance company. For more information, see Chapter 14, "The Insurance Claim Processing Application."
HelloWorld	Demonstrates how to use a custom activity in an application. The custom activity displays a greeting based on the language specified for the greeting. For more information, see Chapter 17, "Writing Custom Activities."
TimeOffRequest	Submits an employee's request for time off for approval by the manager and by human resources (HR).

You can use these applications as models for your own application. To base an application on one of these existing applications, use Save As and make sure your new application has a unique name and a unique database table.

After configuring a sample application, you can deploy it in development mode to your own cluster. In this way, you can see how applications are managed in Process Administrator. You can also see how end users run them in Process Express.

Applications and the Corporate Directory

When you install PAE, the default installation uses the same Directory Server for both the configuration directory and the corporate user directory. This user directory contains a small set of sample users, along with users defined as administrative users during the PAE installation. For example, if you define `admin` as the administrative user, the installer adds this to the sample set of users in the directory.

If you do a default installation, using the `admin` user, you can easily set up the sample applications to deploy.

If you do a custom install or otherwise change your configuration, you may need to make adjustments to the samples before you can deploy them. For example, if you defined a different user directory (such as your company's own corporate directory) or if you want to use other users, you must adjust the sample applications.

Most of the sample applications define `admin` as their default user. If you use your company's corporate directory, then you must do either of the following before deploying the sample applications:

- Make sure your company's corporate directory contains an `admin` user.
- Use another user ID, such as your own or a test user ID.

If you change your corporate directory after your cluster has been created, there are several changes that you and the server administrator must make before you can deploy applications with the new directory. For information on changing your user directory, see the *Administrator's Guide*.

Setting Your Corporate Directory

Before you can add or change the default users for the sample applications, you must define a corporate directory for each application. Otherwise, you cannot browse or search the directory of users. To define a directory, perform the following steps:

1. Open the application.
2. In the application tree view, highlight the name of the application, right-click, and choose Properties.
3. Click the right arrow in the Corporate Directory property field. A selection dialog box appears for the corporate user directory.
4. Choose whether to set the directory based on the cluster or not.
 - If you have a cluster available during the design phase, choose the cluster-based directory, and select the cluster you want.
 - If you don't have access to a cluster, choose to point to a specific directory. A drop-down list appears, listing all corporate directories identified in the `preferences.ini` file. Note that if you point to a specific directory, you may need to change this value when you deploy the application. This will be necessary if your deployment cluster uses a corporate directory different from the one you chose.
5. Click OK.

Deleting an Application

You can delete applications only when they are stored on your local machine. To delete an application, perform the following steps:

1. In the Select an Application window, select an application.

If this window is not open, you can open it as follows. From the Application menu, choose Delete.

2. In the Select an Application window, click the Delete button.

A confirmation dialog box appears.

3. Click Yes to delete the application.

Designing a Process Map

This chapter describes the elements of a process map, their properties, and how to use them to design a process map.

This chapter includes the following sections:

- Drawing the Process Map
- Saving a Process Map to a File
- Adding Items with the Palette
- Deleting Items
- Entry Points
- User Activities
- Automated Activities
- Subprocesses
- Custom Activities
- Exception Manager
- Decision Points
- Split-Join (Parallel Processing)

- Notifications
- Exit Points
- Transitions

Drawing the Process Map

When you create a new application, Process Builder displays an initial set of empty resource folders and a blank process map. As you draw the map, Process Builder adds items to the folders in the application tree view.

To design a process map, follow these steps:

1. Create a new application.
2. Drag items from the Palette to the Process Map pane.
3. Update the properties of the items you add to the process map.
4. Add transitions between the items.

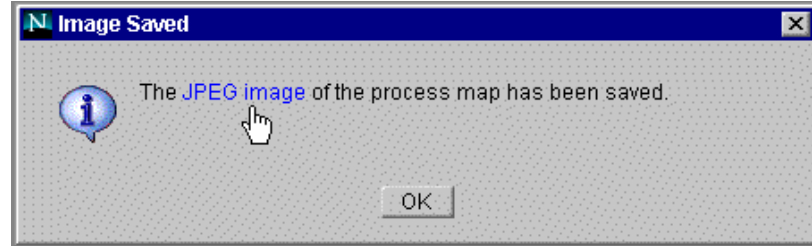
A process map is the visual representation of an application's steps. You also need to create all the other items, forms, data fields, and so on, in order to make a working application.

Saving a Process Map to a File

Sometimes you want to export the process map—in other words, save the map outside of Process Builder. For example, by saving the process map as an image, you can later print it or insert it into an HTML document.

To export the process map, open the Application menu and choose “Save Process Map as JPEG.” This menu item saves the process map as *application_name.jpg* in the folder of the application. After the process map is saved, a dialog box appears, as shown in Figure 4.1. You can click OK to close this dialog box, or you can click “JPEG image” to display the new image file that was created.

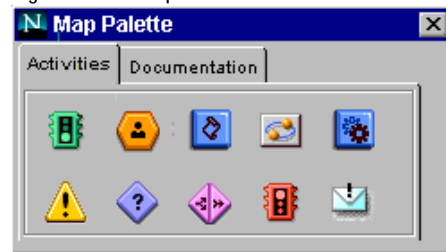
Figure 4.1 After saving the process map, you can view it by clicking “JPEG image”



Adding Items with the Palette

You create a process map by dragging items from the palette (shown in Figure 4.2) to the process map.

Figure 4.2 The palette



Using the palette, you can add the map items described in Table 4.1

Table 4.1 Icons in the Activities tab











Icon	Description
	Entry Point. A point at which a user can initiate a process. An application can have several entry points. For example, if the first few steps create an ID number for the user, a returning user who already has an ID can skip those steps.
	User Activity. A step within the process that requires a user to perform a task. Each user activity has an assigned user who performs the task (assignee) and a form the user needs to fill out in Process Express. After you place activities in the process map, you define the sequence in which they are to be executed by connecting them with transition lines.

Table 4.1 Icons in the Activities tab

Icon	Description
	Automated Activity. An automated step performed through a JavaScript script without user intervention.
	Subprocess. A step that calls a process from within another process. The process that calls the subprocess is considered to be the main, or parent process, and the subprocess is considered to be its subordinate or child process. A parent process can have several children processes, each of which is a stand-alone process complete with entry and exit points.
	Exception Manager. A step that allows the administrator to intervene manually if errors occur when users run the application.
	Decision Point. A conditional step that causes the process to use different steps based on a condition. For example, you might have a decision point that directs the process to different steps depending upon the cost of an item.
	Split-Join (parallel processing). A step within the process that branches in two or more branches so that two or more activities can execute in parallel.
	Exit Point. A step at which the process ends. An application can have several exit points. For example, in a vacation time off request application, an exit point could be the approved vacation request, and another could be a vacation that was not approved.
	Custom Activity. A step at which a PAE application connects to external components or services.
	Notification. An email notification that is triggered when a user activity is started. The email can serve many purposes. For example, it can inform the person who started the process or other users of the process's progress.

To add items to your process map from the Palette, follow these steps:

1. Click the Palette icon in the strip below the menu bar to display the Palette, if it's not displayed already.
2. Drag an item from the palette to the process map.

For notifications, you must drag the item into the user activity or exit point that already exists on the process map.

3. Double-click the item or highlight it and click the Inspector icon.
4. In the Inspector dialog box, define the item's properties.

You can also right-click the items in the process map or application tree view. Since notifications do not appear on the process map, you must select them from the application tree view. For more information on the properties of specific items, see the section in this chapter for the type of item you added.

Deleting Items

To delete a process map item, you can do one of the following in the application tree view or the process map:

- highlight the item and press the Delete key
- right click the item and choose Delete from the menu
- highlight the item from the Edit menu and choose Delete

When you delete an item that is connected to other items by transitions (actions), you must also delete the transitions. If you attempt to delete an item with transitions, a dialog box appears asking if you also want to delete the transitions.

Entry Points

An entry point is a point at which a user can create a process instance. You can have more than one entry point; for example, if you are designing an application to create a data sheet, you might have two entry points. One entry point might be for data sheets for which you have to create art. Another entry point, later in the process, might be for data sheets for which you already have art prepared.

Entry points have the following properties:

Name The name of the entry point that appears in Process Builder and Process Express.

Description An optional longer description of the entry point that appears in Process Express.

Completion Script A script that runs when the step is completed. You must have already written the completion script you want before filling in this property.

User Activities

A user activity is a task that must be performed by a participant (the assignee) as part of a process. For example, a manager may have to approve an employee's time off. A form associated with the activity enables the assignee to take the required action so that the process moves to the next step.

User activities have the following properties:

Name The name of the activity that appears in Process Builder, and in Process Express. Since this name appears on users' work lists, make it descriptive of the action they need to take.

Description An optional longer description of the activity that appears in Process Express.

Allow to Save True/false. When set to true, this option allows users to save a work item so that they can complete it at a later time. If you set this property to true, a Save button appears on the work item's form in Process Express. Defaults to false

Allow to Add Comment True/false. When you set this option to true, end users are allowed to add comments to the activity. The comment field appears automatically on forms presented to users for that activity in Process Express. The assignee can add comments, which are then displayed in Process Express as part of the process instance's history. Defaults to true.

Allow to Delegate True/false. When set to true this option allows users to delegate a step so that someone else can handle it for them. If a step is delegated, the person it is delegated to becomes the step assignee, with the associated access to forms. If you set this property to true, a "Delegate" button appears on the work item's form in Process Express. Defaults to false.

Assignment Script The script that assigns a user or group of users to a work item. When a user is assigned a work item, it appears in the user's work list in Process Express. See "Setting Activity Assignments" on page 81 for further details about the script properties. You can use a predefined assignment script or create your own. If you create your own, the script must already exist before you can enter it in this property.

Expiration Setter Script The script that sets an expiration date or time for a work item. See "Setting Activity Expirations" on page 79 for further details about the script properties. The script must already exist before you can enter it in this property.

Expiration Handler Script The script that runs when a work item has not been completed by its expiration date or time. The script must already exist before you can enter it in this property.

Completion Script A script that runs when the step is completed. The completion script must already exist before you can enter it in this property.

Exception Manager If an error occurs, the Exception Manager invokes a new work item for correcting the error. Errors can occur in two cases: if a completion script returns false, or if a runtime error occurs in a script (for example, in an assignment, expiration setter, or expiration handler script). See "Exception Manager" on page 97.

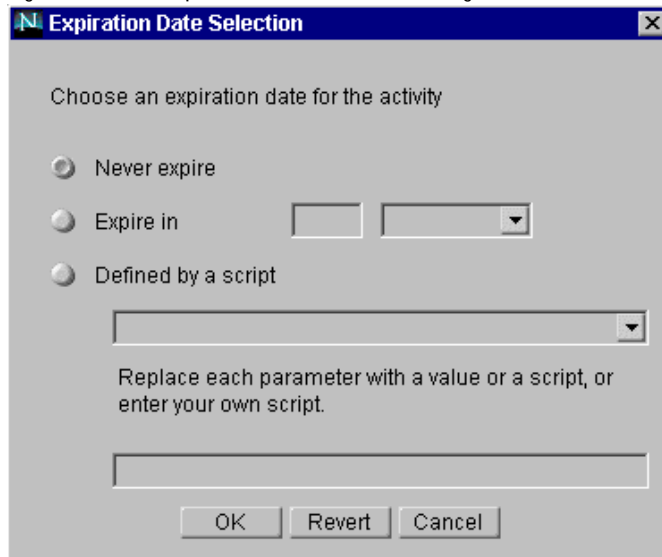
Setting Activity Expirations

When you create a user activity, you can set an expiration date or time for it in the Expiration Date Selection window, either by specifying an expiration or a script to set the expiration. To set an expiration, follow these steps:

1. Open the Inspector Window for an activity by double-clicking the activity or highlighting it and clicking the Inspector icon, or right-click and choose Properties.
2. From the Expiration Setter Script field, click the right arrow button.

The Expiration Date Selection dialog box appears, as shown in Figure 4.3.

Figure 4.3 The Expiration Date Selection dialog box



3. Click the radio button for the type of expiration you want.
4. Click OK to close the dialog box.

The following options are available:

Never expire The activity never expires. This is the default value.

Expire in The activity expires in a specified amount of time. You set the number and a unit of measurement (Months, Weeks, Days, Hours, Minutes).

Defined by a script The script that determines when the activity expires. The product contains no built-in scripts of this type. You can write your own JavaScript scripts, which will then be available in the drop-down list below the Defined by a script radio button.

If you need to set parameters for your script, you define them in the last field in the dialog box. You can give the parameter values, or the names of scripts that return the value. You can also enter your own script here. Enter your own script if you want to use a script that you haven't written yet, or if you want to use a script that is not an expiration setter script (for example, a toolkit script). If you write your own script, it must follow the conventions for expiration setter scripts.

If you set an expiration time or date, you can also set an Expire Manager Script for the activity. An Expire Manager script runs when the activity expires. This script is optional. You must create your own JavaScript script, since none of these scripts are built-in.

Setting Activity Assignments

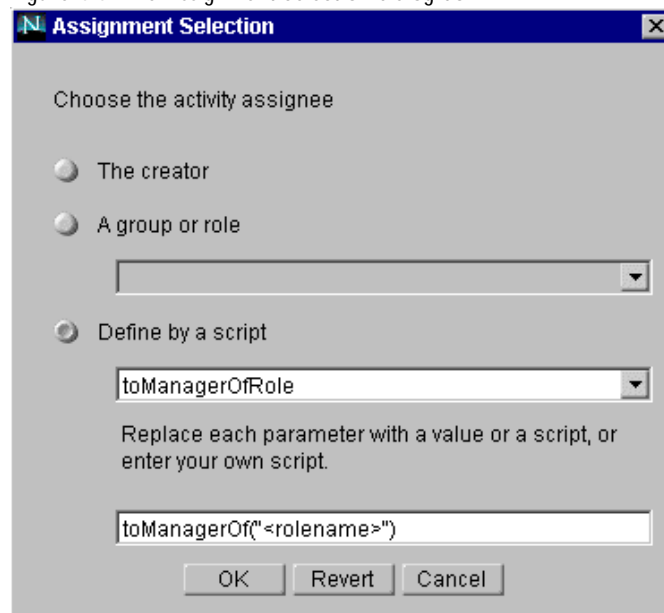
You can set a user to be assigned to an activity through an assignment script, many of which are built into Process Builder. You may need to set up some groups and roles before you can assign activities. See Chapter 5, “Defining Groups and Roles,” for more information.

To assign an activity, follow these steps:

1. Right-click the activity in the process map or application tree view and choose “Set assignee script”. You can also access the property page and click the Assignment Script field.

The Assignment Selection dialog box appears, as shown in Figure 4.4.

Figure 4.4 The Assignment Selection dialog box



2. Click the radio button beside the type of assignment you want.
3. Click OK to close the dialog box.

There are three ways to specify an assignment:

The creator Assigns this activity to the creator (the person who initiated the process instance).

A group or a role Select one of the roles or groups in the drop-down list. The list contains all the default roles and groups, as well as the ones you've created. If you assign a task to a group, the person in the group who first accepts the work item in Process Express performs the task. The other members of the group do not perform the task. If you want to assign a task to every member of a group for approval, see "Using Parallel Approval" on page 82.

Defined by a script Assigns this activity using a script—either a built-in script specified in the drop-down list, or a script whose name you specify in the text field below the radio button.

If you select the "Defined by a script" radio button, you must supply values for parameters contained in angle brackets. For example, using the assignment script `toManagerOfRole`, you are asked to complete the parameter in `toManagerOfRole("<rolename>")`. You must replace `<rolename>` with the field role: for example, `toManagerOfRole("product_mgr")`. You can also replace parameters with scripts that return the parameter value.

For details on the available assignment scripts you can specify, see "Assignment Scripts" on page 181.

Using Parallel Approval

Usually only one user is assigned an activity. However, in some cases you might want to have several users perform the same activity consecutively, for example, if you want several people to approve a document or action before it moves on to the next step. To implement an activity of this sort, you use parallel approval.

You implement parallel approval by creating an activity with specific assignment and completion scripts, and a data field to store necessary information.

To set up a parallel approval activity, follow these steps:

1. Create a data field *fieldname* to keep track of who has already performed the review and who still needs to perform it. This field must be a computed field with length of 2000.
2. Add an activity to the process map.
3. Right-click the activity in the application tree view and choose Properties to go to the Properties window for this activity (which will be your parallel approval activity).
4. Click the arrow on the Assignment Script property to bring up the window for setting the assignment script.
5. Choose `toParallelApproval` from the drop-down list.
6. Replace the parameters *arrayOfUserDNs* and *fieldname* with appropriate values.
7. Click OK to return to the activity's Properties window.
8. Click the arrow on the Completion Script property to bring up the window for setting the completion script.
9. From the drop-down list, choose `checkParallelApproval`.
10. Replace the parameters *fieldName* and *lableOfStopperAction* with appropriate values, as described in the next section.
11. Click OK.
12. Fill in any other properties you need to for the activity.
13. Close the window. Your changes are saved automatically.

From the end user point of view, a parallel approval item is assigned to all users in the group that need to give approval. If a user starts the work item and approves it, PAE stores the result in the data field and reassigns the work item to every user but the user who has approved the work item. This continues until all users have approved the item, or until one of the users has not

approved the item. If all users have approved the item, the process continues in one direction. If one of the users has not approved the item, the process continues in another direction.

Parallel Approval Completion Script

To use parallel approval, you must use the completion script `checkParallelApproval(fieldName, labelOfStopperAction)`.

This script runs when the parallel approval activity is completed. If any user chooses the “stopper action” (that is, refuses to approve the item) the completion script performs the appropriate action. If all users complete the activity without choosing the “stopper action” (that is, all approve the item) this script performs the appropriate action.

The parameters are `fieldName` and `labelOfStopperAction`.

- The `fieldName` is the field that keeps track of who has performed the approval and who still needs to do so. The field is a computed field of length 2000 that you have to add to the data dictionary.
- The `labelOfStopperAction` is the name of the action or transition that a user can select that stops the approval.

Automated Activities

Automated activities are steps performed by PAE without user intervention, through one or more JavaScript scripts. Automated activities are triggered as soon as the process instance reaches the activity, unless the activity is deferred. If the activity is deferred, it is triggered at its specified date and time.

Automated activities have the following properties:

Name The name of the automated activity that appears in Process Builder.

Description An optional longer description of the activity.

Schedule The schedule for deferred activity. You need to set the `Deferred` property to true to use this schedule. You set the `schedule` property in a pop-up dialog box, shown in Figure 4.5.

Figure 4.5 Automated Activity Schedule dialog box

Set the schedule for this automated activity.
The format is HHMM, using 24-hours clock. Separate multiple times with commas (0620,2030)

Sunday

Monday

Tuesday

Wednesday

Thursday

Friday

Saturday

OK Revert Cancel

Enter a time into any of the day fields to have the automated activity run at that time on that date. Enter times in the 24-hour clock format (for example, 1300 for 1:00 pm). To enter more than one time for a day, separate the times with a comma. If you enter only two digits, Process Builder assumes they are the hour.

Deferred This property specifies if the activity should be performed as soon as the process reaches the automated activity, or if it should be deferred. If you set the property to true, the automated activity is deferred to the time you specify in the Schedule property. If this property is set to False, the automated activity is performed immediately. The default is false.

Automation Script The script that performs the automated step. The script must already exist before you can enter it in this property.

Completion Script A script that runs when the step is completed. The script must already exist before you can enter it in this property. For automated activities, the step is completed when the automation script has finished running, so the completion script runs immediately after the automation script.

An automated activity also has a Transition Order window, where you can rearrange the possible transitions when there is more than one. You display the Transition Order window by clicking the Transitions tab in the Inspector window.

Transitions out of automated activities always default to true, because a true outcome means the process instance moves on to the transition. When you have more than one transition out of an automated activity, the activity operates in the same way as a decision point, which evaluates multiple conditions (for example, if the price is over \$100 or less than or equal to \$100) in a specific order. You set the order in the Transition Order window, by clicking the up and down arrows. The top condition in the window is evaluated first. PAE uses the first transition condition that is true to route the process instance.

Exception Manager If an error occurs, the Exception Manager invokes a new work item for correcting the error. Errors can occur in two cases: if a completion script returns false, or if a runtime error occurs in a script (for example, in an assignment, expiration setter, or expiration handler script). See “Exception Manager” on page 97.

Subprocesses

A subprocess is a fully functional process that is called from within another process. The process that calls the subprocess is considered to be the parent process and the subprocess is considered to be its child process. A parent process can have several children processes, each of which is a stand-alone process complete with entry and exit points.

Subprocesses allow process designers to reuse processes across their environment. For example, if a bank has many different loan management processes, all of which require a credit check at a certain point in the process, each of the loan management processes can use the same credit check subprocess. Chapter 13, “The Loan Management and Credit History Applications,” describes the Loan Management sample application and its subprocess the Credit History application; the two sample applications show how to use a subprocess in your applications.

Another advantage of subprocesses is that the logic flow of the main process map may be simpler to follow because a certain portion of its processing is hidden behind the subprocess icon. For example, in the Loan Management

sample application, the Check Credit History processing is handled in the Credit History subprocess and is not exposed in the main Loan Management process map.

You can use a subprocess in the same way as an automated activity (see “Automated Activities” on page 84). You can place it anywhere on the process map, have multiple transitions coming out of it, and place complex scripts within the subprocess to perform actions that are hidden from the main process.

A timer agent coordinates how the parent and child process interact together. The parent process initiates a child process; when the child completes, it sets a value, `wf_observer_url`, in a Process database table. This is the URL of the parent process. A timer agent periodically checks this database table (by default, every five minutes). If there’s a value in the `wf_observer_url` field, the timer agent attempts to connect the child to the parent process. Once the child successfully connects back to the parent, the value in the database table is set to null so the timer agent does not attempt to reconnect again.

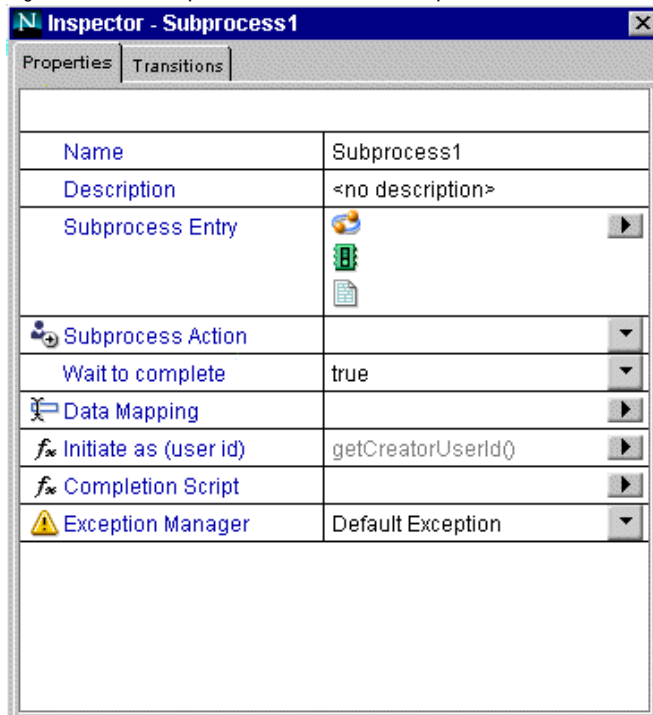
If there’s a problem in connecting to the child or to the parent, an exception is thrown and the exception manager allows the administrator to intervene. See “Exception Manager” on page 97 for more information.

To set up a subprocess, follow these steps:

1. Create a child process.
2. Open the parent process map.
3. Drag a subprocess icon onto the parent process map.
4. Right-click the subprocess icon and choose Properties.

The Subprocess Inspector Window appears, as shown in Figure 4.6.

Figure 4.6 The Inspector Window for a subprocess



5. Fill in the properties for the subprocess.
6. Close the window. Your changes are saved automatically.

All subprocesses have the following properties:

Name The name of the subprocess.

Description An optional longer description of the subprocess.

Subprocess Entry The entry point in which the parent process enters the child process. To enter the subprocess entry:

1. Click the right arrow next to the subprocess icon. The Subprocess Entry Chooser window appears.
2. In the list of applications, click the plus (+) sign of the application you want to make a subprocess.

3. Choose an action or an entry point and click OK.

Subprocess Action The action that you follow from the Subprocess Entry field.

Data Mapping The data that must be mapped between the child and parent process so that values can be passed easily between them. To define the data mapping:

1. Click the right arrow in the Data Mapping field. The Data Mapping Setting window appears.
2. In the Mapping Script column, click the arrow for each row. The Field Mapping window appears.
3. Enter a static value, select a corresponding field from the parent data dictionary, or define a script and click OK.
4. When you have mapped all the data fields, click OK.

Initiator User Id The initiator of the subprocess.

You can use a dynamically derived value as the initiator. For example, in a hiring application that uses a child process to set up a secure ID card for each new hire, you identify a different new employee in each instance of the parent process and you want to use the new user IDs in the child process. In this case, you use a script to pull the new user ID from the corporate directory.

You can use the App User Id of the parent process as the creator of the child process. In this case, you leave the Initiator User Id blank because the default is to use the same user as the creator of the main process.

You can also use a static value as the initiator, which is especially useful when testing. The sample application, for example, uses a static value (“admin”) to simplify setting up and deploying the application.

Completion Script The script that runs when the child process is completed. The script must already exist before you can enter it in this property.

Exception Manager If an error occurs, the Exception Manager invokes a new work item for correcting the error. Errors can occur in two cases: if a completion script returns false, or if a runtime error occurs in a script (for example, in an assignment, expiration setter, or expiration handler script). See “Exception Manager” on page 97.

Connecting the Parent and Child Process

When a parent process arrives at a child activity, it tries to connect to the subprocess and launch it. To do this, the parent process must have a user ID that can be authenticated with the subprocess.

To connect the parent and child process:

1. Open the parent process map.
2. Right-click the application in the application tree view and choose Properties to go to the Properties window.
3. In the App User ID field, enter the name of the trusted user that can access the child process in order to set the initiator of the child process to match the Initiator User ID in the Subprocess Properties window.
4. In the App User Password field, enter the user's password.
5. Close the window. Your changes are saved automatically.

You may also want to add the App UserID of the parent process to the trusted user group of the child process so the parent process can redefine the initiator of the child process:

1. Open the child process.
2. Add a trusted user to the child process. For more information about adding a trusted user, see “Creating Groups and Roles” on page 114.
3. Close the window. Your changes are saved automatically.

Custom Activities

Custom activities are Java classes created by a developer that you can use in Process Builder applications to perform complex automated tasks, such as connecting PAE applications to external applications and databases.

Custom activities are similar to automated activities, in that they perform tasks without user intervention. However, custom activities, because they are Java classes, can be used for more complicated tasks than the JavaScript scripts of automated activities.

This section presents an overview of custom activities. For a detailed example, see Chapter 17, “Writing Custom Activities.”

Using a Custom Activity

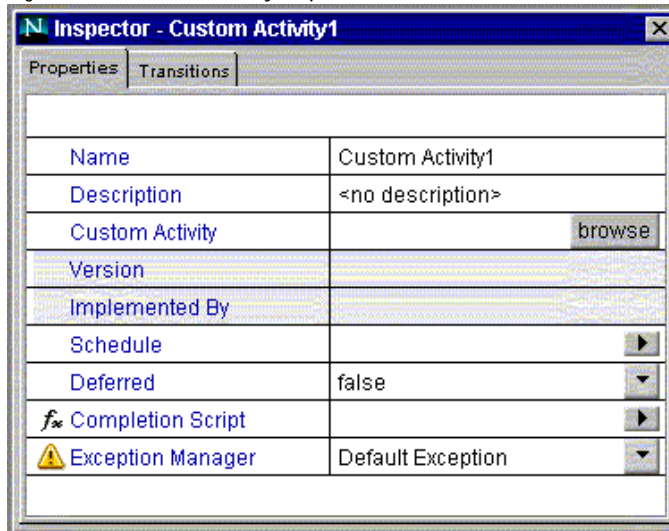
After you add a custom activity to the palette, you can use the custom activity in your process map. To do so, perform these main steps:

1. Drag the custom activity icon from the palette to the process map.
2. Right-click the custom activity icon on the process map and choose Properties. The Inspector Window appears.
3. Click the Input and Output tabs to make sure the data is mapped correctly between the custom activity and the application’s fields.

Custom Activity Inspector Window

Figure 4.7 shows the Inspector window for a custom activity when you first drag the activity to the process map.

Figure 4.7 Custom Activity Inspector



Note that there are only two tabs in the window: Properties and Transitions. The Properties tab displays the following properties:

Name The name of the custom activity that appears in Process Builder, and in Process Express. Since this name appears on users' work lists, make it descriptive of the action they need to take.

Description An optional longer description of the activity that appears in Process Express.

Custom Activity The file that contains the Java class and its xml descriptor. Use Browse to go to an .xml, .zip, or .jar file to use. After you set this property, the Inspector window changes to reflect information defined by the custom activity you selected.

Version The custom activity's version number.

Implemented By The Java class that implements this custom activity.

Schedule The schedule for deferred activity. You need to set the Deferred property to true to use this schedule. You set the schedule property in a pop-up dialog box, shown in Figure 4.8.

Figure 4.8 Custom Activity Schedule dialog box

Automated Activity Schedule

Set the schedule for this automated activity.
The format is HHMM, using 24-hours clock. Separate multiple times with commas (0620,2030)

Sunday

Monday

Tuesday

Wednesday

Thursday

Friday

Saturday

OK Revert Cancel

Enter a time into any of the day fields to have the custom activity run at that time on that date. Enter times in the 24-hour clock format (for example, 1300 for 1:00 pm). To enter more than one time for a day, separate the times with a comma. If you only enter two digits, Process Builder assumes they are the hour.

Deferred This property specifies if the activity should be performed as soon as the process reaches the custom activity, or if it should be deferred. If you set the property to true, the custom activity is deferred to the time you specify in the Schedule property. If this property is set to false (the default), the activity is performed immediately.

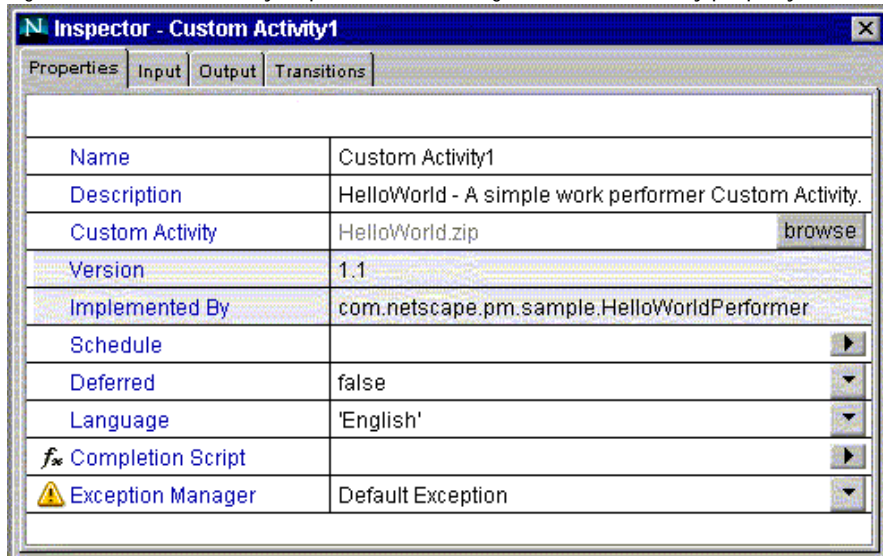
Completion Script A script that runs when the step is completed. The script must already exist before you supply it as a property. The completion script runs immediately after the custom activity finishes executing.

Exception Manager If an error occurs, the Exception Manager invokes a new work item for correcting the error. Errors can occur in two cases: if a completion script returns false, or if a runtime error occurs in a script (for example, in an assignment, expiration setter, or expiration handler script). See “Exception Manager” on page 97.

Inspector Window After Setting a Custom Activity

Once you set the Custom Activity field, the Inspector window changes. An example is shown in Figure 4.9:

Figure 4.9 Custom Activity Inspector, after setting the Custom Activity property

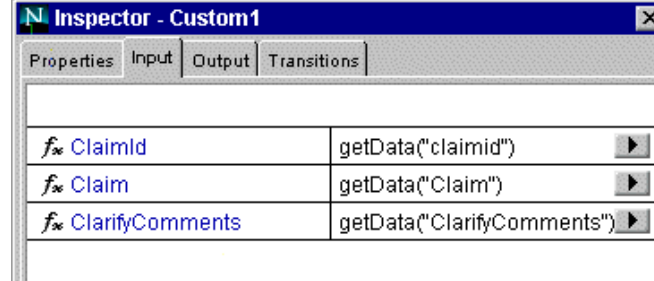


After setting the custom activity, if an environment parameter is set in the custom activity file, the parameter appears in the properties window. In Figure 4.9, Language is an environment parameter.

Two new tabs also appear: Input and Output.

The Input tab shows the parameter names in the custom activity's input hashtable, and shows how the parameter value is derived. Figure 4.10 shows an example of the Input tab.

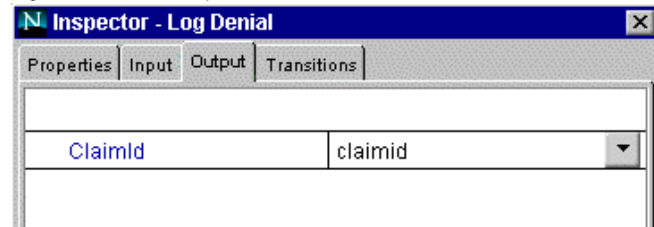
Figure 4.10 The Input tab



For example, the value for the input parameter `claimId` comes from getting the value in the application's `claimid` data field.

The Output tab shows the mapping between the output parameters in your custom activity and the data field in your application. Figure 4.11 shows an example of the Output tab.

Figure 4.11 The Output tab



In this example, the value from the output parameter, `claimId`, is put into the application's data field `claimid`.

Adding a Custom Palette

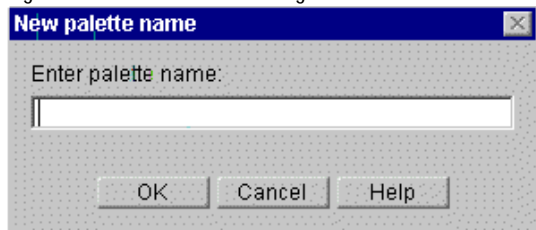
If you frequently use custom activities, you can add them to your palette, thereby giving you easy access to ready-made components.

To add a custom palette, follow these steps:

1. Right click on the Map Palette window and choose "Add custom palette."

A dialog box appears, as shown in Figure 4.12:

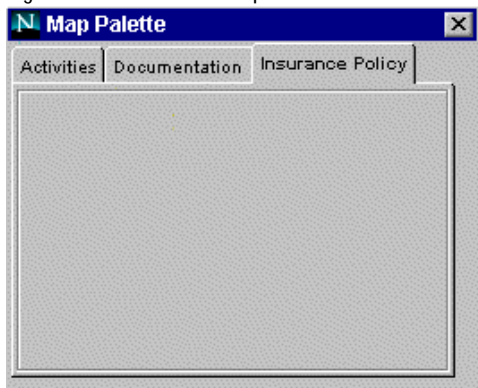
Figure 4.12 New Palette Dialog Box



2. Enter the name of the palette and click OK

A tab containing the custom palette is added to the Map Palette, as shown in Figure 4.13.

Figure 4.13 Blank custom palette



3. Add an item to the custom palette by right-clicking on the custom palette and choosing "Add custom activity."
4. Browse to the .xml, .jar, or .zip file that contains your custom activity.

An icon representing the custom activity appears on the custom palette, as shown in Figure 4.14.

Figure 4.14 Custom palette with custom activity



5. To add the custom activity to your application, drag the icon from the custom palette to the process map.

For information about writing the XML and Java code for a custom activity, see Chapter 17, "Writing Custom Activities."

Exception Manager

Exception managers allow users to intervene if errors occur in the application. You create an exception manager by dragging the exception manager icon to the process map. Most process map items represent steps in the process and are connected by transitions. Exception managers, however, are usually placed off to the side on process maps. They are not connected to other steps; instead, they are called by activities when an error occurs. You set up which exception manager is used at each activity using the Exception Manager property in the activity.

If a problem occurs while the work item is being processed, the exception manager assigned to that activity is called. The exception manager then generates a work item based on the exception manager properties. The work item is assigned to a user by the assignment script. Often the person assigned the work item is an administrator or the creator of the process instance, because these people are in a good position to diagnose what went wrong with the process.

You can attach a notification to the exception manager, which sends mail to the administrator when an exception is thrown. See “Notifications” on page 104 for more information.

Default Exception Manager

Every new process automatically has a default exception manager that appears in the application as soon as you create an activity that requires an exception manager. When you create new process map items that have Exception Manager properties, the properties default to the default exception manager.

To see the properties of the default exception manager, right-click the default exception manager in the application tree view and choose Properties. The default exception assigns the exception work item to the creator of the process instance. You can change the properties of the default exception handler, or you can create your own exception handler.

Creating an Exception Manager

To add your own an exception manager, follow these steps:

1. Drag an exception manager icon from the map palette onto the process map.
2. Right-click the exception manager icon and choose Properties. Update any properties as needed.
3. Assign this exception manager to a user activity, custom activity, automated activity, subprocess activity or end point.

Right-click the activity icon and choose Properties to go to the Properties window. In the Exception Manager field, choose the exception manager from the drop-down list.

4. Either create a new form for the exception manager, or use an existing form. Because exception handlers don't have transitions, the button names are assigned automatically: Retry and Continue.

Retry resubmits the work item where the error occurred. Continue skips the step in the process instance that caused the error. This option should be used with caution.

5. Assign the form to the exception manager using the Form Access window.

Exception Manager Properties

All exception managers have the following properties:

Name The name of the exception manager.

Description An optional longer description of the exception manager.

Allow to Save True/false. When set to true, this option allows users to save a work item so that they can complete it at a later time. If you set this property to true, a Save button appears on the work item's in Process Express. Defaults to false.

Allow to Add Comment True/false. When you set this option to true, end users are allowed to add comments to the activity. The comment field appears automatically on forms presented to users for that activity in Process Express. The assignee can add comments, which are then displayed in Process Express as part of the process instance's history. Defaults to true.

Allow to Delegate True/false. When set to true this option allows users to delegate a step so that someone else can handle it for them. If a step is delegated, the person it is delegated to becomes the step assignee, with the associated access to forms. If you set this property to true, a "Delegate" button appears on the work item's form in Process Express. Defaults to false.

Assignment Script The script that assigns a user or group of users to a work item. When a user is assigned a work item, it appears in the user's work list in Process Express. You can use a predefined assignment script or create your own. If you create your own, the script must already exist before you can enter it in this property. See the programming information in this book for more information on creating scripts.

Expiration Setter Script The script that sets an expiration date or time for a work item. See "Setting Activity Expirations" on page 79 for further details about the script properties. The script must already exist before you can enter it in this property.

Expiration Handler Script The script that runs when a work item has not been completed by its expiration date or time. The script must already exist before you can enter it in this property.

Completion Script A script that runs when the step is completed. The completion script must already exist before you can enter it in this property.

Decision Points

Decision points are places where the process branches into different steps depending upon conditions. For example, if a price is above a certain amount, the process might branch to include an approval step that is unnecessary if the price is below a that amount.

Decision points have the following properties:

Name The name of the decision point that appears in Process Builder.

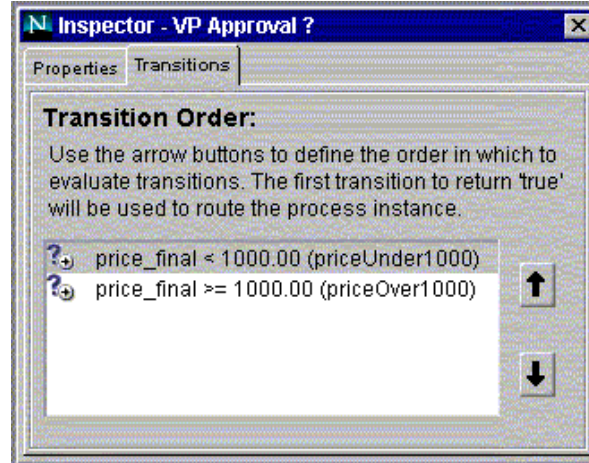
Description An optional longer description of the decision point.

Completion Script A script that runs when the step is completed. The script must already exist before you can enter it in this property.

You add possible outcomes to the decision point by adding transitions with conditions. For example, you could create two transitions, one if the price stored in the `price_final` field is over \$1000, and another if the price is less than or equal to \$1000. Make sure that all possible outcomes are covered in your transitions. For example, if you made transitions with the conditions that the price had to be greater than \$1000 or less than \$1000, you have not covered the condition of the price being equal to \$1000. If you do not have options that cover all cases, the transaction rolls back and the user receives an error message.

Decision points also have a Transition Order window, where you can rearrange the transitions leading from the decision point. You display the Transition Order window by clicking the Transitions tab at the top of the Inspector window. An example from the DataSheet sample application is shown in Figure 4.15:

Figure 4.15 The decision point Transition Order window



The top item in the window is evaluated first, but you can rearrange items by using the arrow icons. PAE uses the first transition condition that is true to route the process instance. So for the example above, the process checks first to see if the `price_final` field is less than 1000.00. If that is true, the process continues with that transition. If the `price_final` field is more than 1000.00, the first condition is false, so the second condition is now evaluated. If the second condition is true, the process continues with that transition.

For more information on transitions, see “Transitions” on page 106.

Split-Join (Parallel Processing)

Parallel processing allows a process to branch in two or more directions so that two or more activities can execute in parallel. In complex processes, you can nest parallel processes within a larger parallel process. Note that the activities in the nested process are considered to be part of the nested process, not the larger process; that is, if you nest process “a” within process “A”, each with its own pair of split-join icons, the activities in the nested process must progress from split “a” to join “a” not to join “A.”

Chapter 12, “The Office Setup Application,” describes the Office Setup sample application, which controls the process of setting up an office for a new employee. In this sample application, each subtask is grouped into a processing branch that progresses independently of the other subtasks. For example, the

MIS department can set up the phone while the purchasing department is ordering the computer. Problems completing one task won't affect the progress of a parallel task.

Properties of a Parallel Process

Parallel processes have the following properties:

Name The name of the split or join.

Description An optional longer description of the split or join.

Completion Script A script that runs when the step is completed. The script must already exist before you can enter it in this property.

Parallel processing also has evaluation order windows, where you can rearrange the transitions leading from the split or join icons. You reach the evaluation order window by clicking the evaluation order tab at the top of the inspector window. The top item in the evaluation order window is evaluated first. Rearrange items using the arrow icons.

Adding a Parallel Process

To use parallel processing in your application, follow these steps:

1. Drag a split-join icon onto the process map.
2. Between the split and the join icons, add the desired activities.
3. Drag transitions from the split to the activities. The transitions are similar to the conditions out of decision points and default to a value of true. For more information, see "Decision Points" on page 100.
4. You can reorder the evaluation order of the conditions as you would for conditions out of a decision point. For more information, see "Decision Points" on page 100.
5. Drag transitions from each of the activities in all branches from a split so that they eventually end up in the corresponding join.

The rules for using parallel processing activities are as follows:

- A branch (or thread) of activities out of a split must eventually complete in its corresponding join. Activities that are part of a processing branch out of a given split must progress through that branch until the corresponding join.
 - They cannot exit out of the processing branch directly to an exit point.
 - They cannot transition out of the processing branch to an activity that is part of another processing branch.
 - They cannot transition out of the processing branch to an activity that is part of a subordinate nested parallel process, which would be therefore part of another processing branch.
 - If they are in a nested parallel process within a larger, higher-level parallel process, they cannot transition to an activity that is in the higher-level parallel process.
 - They cannot transition back into the split icon that started the processing branch they are a part of.
 - They cannot transition back to the activity that precedes the split icon.
- If you delete a split or a join, the corresponding half icon is also deleted (you must also delete any transitions in or out of a split or join at the same time).
- After you deploy an application to test or production, you cannot delete a split or a join.
- You cannot loop on a split or a join.
- You can have conditions set on transitions that come out of a join. However, if the join fails, then the transaction rolls back to the most recent activity in the process block. Similarly, if no condition out of a split evaluates to true, then the process rolls back to the most recent activity.
- By default, when you first create a new split or join, the other half icon is given the same name. You can change this later to display different names.

Notifications

Notifications are email messages sent when the process reaches an activity. One use of notifications is to notify the person who needs to perform the next step in the process. Another use is to give a status (for example, information about the document's progress within the process) to the person who created the document. You can attach notifications only to user activities and end points.

When the end user receives a notification, the title of the notification is the application's title field, followed by the priority field. For more information on these fields, see "The Application Properties Dialog Box" on page 66.

Notification Properties

Notifications have the following properties:

Name The name of the notification that appears in Process Builder.

Description An optional longer description of the notification. For example, it could describe the step in the application where the notification happens.

Email Address(es) The email address for the notification. This can be a static email address, but more commonly would be a script that assigns an email address based on the instance of the process. For example, a script like `emailOfCreator` sends a notification to the email address of the person who created the process instance. If you want to use a static email address, you must put quotation marks (" ") around it. Multiple addresses are separated by commas.

Content Type The format the email message's content is in. You can choose between `text/html` (messages sent in HTML format) and `text/plain` (messages sent in plain text format). If your mail system does not support HTML mail, choose `text/plain`, or your notifications may not work. In both cases, the character set is `us-ascii`.

Email Body The content of the email notification you want to send. If you enter static text, you must put quotation marks (" ") around it. You can also enter the name of a script that returns the email message string.

Built-in Email Notification Scripts

As shown in Table 4.2, Process Builder provides built-in scripts that you can use for addressing notifications.

Table 4.2 Built-in Email Scripts

Script	Definition
<code>emailOfRole(roleName)</code>	Sends email to the user who is performing a field role for a process instance. <code>roleName</code> is the name of the field role.
<code>emailOfAssignees()</code>	Sends email to the user assigned to the process step.
<code>emailOfCreator()</code>	Sends email to the creator of the process instance.
<code>emailByDN(userDN)</code>	Sends email to the specified user DN (distinguished name).
<code>emailById(userID)</code>	Sends email to the specified user ID

Exit Points

Exit points are steps at which a user can exit a PAE process. An application can have several exit points. For example, a vacation request process might end with the vacation being approved or denied.

Exit points have the following properties:

Name The name of the exit point that appears in Process Builder.

Description An optional longer description of the exit point.

Exception Manager If an error occurs, the Exception Manager invokes a new work item for correcting the error. Errors can occur in two cases: if a completion script returns false, or if a runtime error occurs in a script (for example, in an assignment, expiration setter, or expiration handler script). See “Exception Manager” on page 97.

Transitions

After you place items from the Palette on your process map, you must connect them to show how data flows or how actions flow. You connect steps with transition lines that have directional arrows indicating their destination item.

Note You cannot use transitions with exception handling. For more information, see “Exception Manager” on page 97.

Types of Transitions

There are two basic types of transitions:

- regular transitions (which do not depend upon conditions and are represented on the process map as blue lines)
- conditional transitions (which depend upon conditions and are represented on the process map as green lines)

Regular transitions originate from activities and do not have a property where you can set conditions for their use. If you have multiple regular transitions from an activity, the process branches. The user chooses between transitions them by selecting a button on a form.

Conditional transitions originate from automated activities and decision points. They are not executed until a condition is met. If you have multiple conditional transitions, they are executed based on evaluation order and condition. PAE evaluates the conditions in the order specified and the first condition of a transition that it finds to be true is the transition it executes.

Note In addition to continuing on to the next step in a process, a transition can loop back to a previous step. To draw a looping transition on a process map, drag the arrow icon backwards to a previous step. For example, if at one step in a process a graphic artist produces a graphic, and the next step is manager review of the graphic, two transitions can lead from the manager's approval step. The first transition, if the manager approves the graphic, continues to the next step in the process. The second transition, if the manager does not approve the graphic, loops back to the previous step so that the graphic artist works on the graphic again.

Adding a Transition

To add a transition, follow these steps:

1. From the arrow just beyond the upper-right edge of an icon on the process map, drag the arrow to another icon. Do not release the mouse button until you've moved the arrow to the inside of the destination icon.

An arrow connecting the two steps appears on the process map. The arrows are blue if they begin in an entry point or activity, green if they begin in a decision point or automated activity.

2. Give the transition a descriptive name.

This name appears as a button on the form (unless the transition is from an automated activity or a decision point).

3. Define the transition's properties in the Inspector Window.

Transition Properties

All transitions have the following properties:

Name The transition's name is the name of the button on the form for the activity at from which the transition originates. The name typically describes something about the activity that is the destination of the transition. For example, a line that goes from "Create Graphics" to "Review Datasheet" might be called "Publish for Review" because when the graphics are done, the art department wants to publish the new datasheet for review.

Description An optional longer description of the transition.

Setting the Property for a Virtual Transition

Transitions that lead from a user activity also have the following property:

Virtual Action If set to true, no button associated with this transition will appear on the assignee's HTML form. A common use for a virtual transition is for activities that expire. If you set up an activity so that it expires after a certain amount of time, the expiration handler script might use this virtual transition to advance the process to another step. The advance occurs through JavaScript,

without the involvement of the assignee. For more information on using JavaScript to move from one process step to another, see the `moveTo(activityName)` function in Appendix A, “JavaScript API Reference.”

Setting the Property for a Conditional Transition

Conditional transitions (from automated activities and decision points) also have the following property:

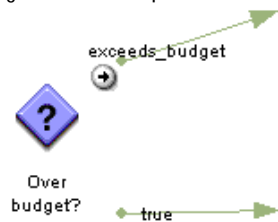
Condition The condition the activity must meet before executing the transition. Conditional transitions can be scripts that return true or false (you can type either the name of the script or type the script itself into the condition property). Conditions can also be based on values of data fields. For example, if your condition is that the price field be over 1000, then your condition can be `price>1000`, where `price` is the price field. If the value of the price field is over 1000, then this condition is true.

The condition defaults to true for automated activities and for custom activities. If there is more than one condition, you can prioritize the conditions in the evaluation order window. For more information on prioritizing conditions, see “Decision Points” on page 100.

Example Using a True/False Field

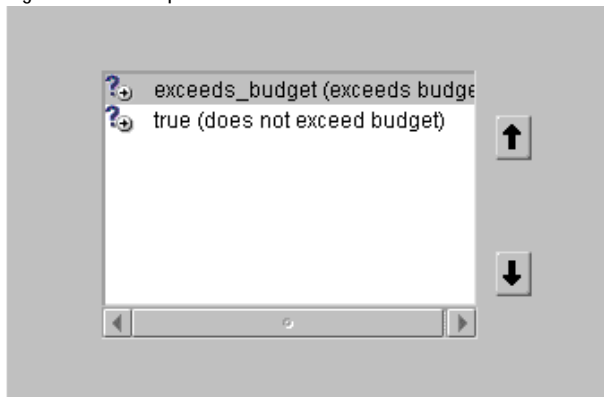
If you have a field with true and false as its valid values, you can use the field name as a transition’s condition. An example is shown in Figure 4.16. Suppose you have a field named `exceeds_budget`, and the value can be true or false. In that case, the first condition is “`exceeds_budget`,” and the second condition is “true.”

Figure 4.16 Example of condition set to a field



To make sure these conditions are evaluated in the proper order, open the decision point's Inspector window and click the Transitions tab. This opens the Transition Order window, as shown in Figure 4.17.

Figure 4.17 Example of the Transition Order window for a decision point



The process evaluates the first condition, "exceeds_budget," to see if the value of the field is true or false. If the value of the field is true, then the condition is true, and the process continues with the "exceeds_budget" transition. If the value of the exceeds_budget field is false, the process evaluates the next condition. In this example, the next condition is true. Therefore, its outcome is always true, but it is evaluated last. A final condition of true works like an "else statement." That is, the true condition automatically advances the process if none of the previous conditions are true.

Defining Groups and Roles

This chapter explains creating groups and roles and defining their properties.

This chapter includes these sections:

- Groups and Roles Overview
- Creating Groups and Roles
- Prioritizing Groups and Roles
- Deleting Groups and Roles

Groups and Roles Overview

For a PAE application, user roles and groups determine which users see which forms and which users perform which tasks. Scripts can also use the groups and roles. Groups are a collection of users in an application. Roles are the parts users play in a specific process instance. For example, you could have a group composed of all employees in the company. If someone in that group submits a time off request, for that particular process instance that employee has the role of creator. The creator role is defined automatically when you create a new application, however, you can create other roles. The user and group information in the corporate user directory is the source of user information when you set up groups and roles.

The following types of groups and roles are available:

Application Group A group defined in a particular application that points to user data in the corporate user directory. The group information is stored in the configuration directory, and the user information is stored in the corporate user directory. Because these groups are stored in the configuration directory, you do not need to have write access to the corporate user directory to create one. The context of this group is the application, since it cannot be used by other applications.

Corporate Group A group that points to a group defined for the corporation in the corporate user directory. The application's corporate group is exactly the same as the group in the corporate user directory; when the group in the corporate user directory is updated, the corporate group automatically uses those changes. Using corporate groups you can leverage the groups already defined in the corporate user directory. The context of this group is in the corporation; it can be used for many things besides applications.

Dynamic Group This group uses a filter to list members in the corporate user directory that match the attributes in the search filter. Unlike a corporate group, which is already defined as a group in the corporate user directory, the dynamic group uses corporate user directory information to create a group dynamically. This group is defined in the context of the corporation (it changes based on changes in the corporate user directory) but it also is used only for the application.

Field Role A role represented as a data field. The value of the field varies by process instance, so the role's context is the particular process instance. For example, the creator role (the person who starts a particular process instance) is a field role. The field role can only be one user; you cannot use a field role to assign an activity to a group.

Internal Group/Role An internal group or role is defined by default in PAE. For example, the all group, which represents all users in the corporate user directory, is an internal group.

There are three high-level steps in creating groups and roles:

1. Choose the type of group or role you want.

You can do this by asking yourself what the context of the group or role is. Is it just for this application, or is it a group that has corporation-wide implications?

2. Choose valid users for the group or role.
3. Prioritize all of the groups and roles in an application.

You need to prioritize the groups and roles so that if a user is assigned to multiple roles, the form access system can determine what precedence to give each role. For more information, see “Prioritizing Groups and Roles” on page 128.

Default Groups and Roles

When you create an application, the following groups and roles are created by default:

all The group of all users in the corporate user directory.

creator The person who initiates a particular process instance.

admin The group of people allowed to administer the application's process instances using Process Business Manager. You need to add the names of the people who can administer the application to the application's default “admin” group. These people must also belong to the WF Admin Auth ACL for the Enterprise Servers. Anyone who is allowed to administer by the WF Admin

Auth ACL can administer the application at the application level, but only users who belong to the application's "admin" group can administer the application at the process instance level.

trusted users A group that can start a subprocess as another user. The parent process redefines the initiator of the child process. If no one will need to start a subprocess as someone else, you need not assign anyone to the trusted users group.

You can customize some aspects of these groups and roles, but you cannot delete them. If you don't need them for your particular application (for example, if your application deals with sensitive information that you do not want to expose to the "all" group) you don't need to define forms for particular groups or roles. See "Setting Access to Forms" on page 172 for more information.

In addition to the above groups and roles, each application has an assignee role. It is set for every step in the process in which someone must perform an action. However, since it is a special role, it doesn't appear in the application tree view and you cannot set properties for it. It works in concert with other groups and roles. For example, a step may be assigned to a group, but the actual user from that group who performs the task in a particular process instance is the assignee.

Creating Groups and Roles

To begin defining a role or group, follow these steps:

1. From the Insert menu, choose Group & Role.
2. In the Create a New Role or Group dialog box, choose the type of role and fill in the fields.

See "The Create a New Role or Group Dialog Box" on page 115 for details.

3. Click either the Add button or the "Add & Define" button.
 - Click Add & Define if you want to define the new item right away. A dialog box for the type of group or role you chose (for example, an application group) appears.

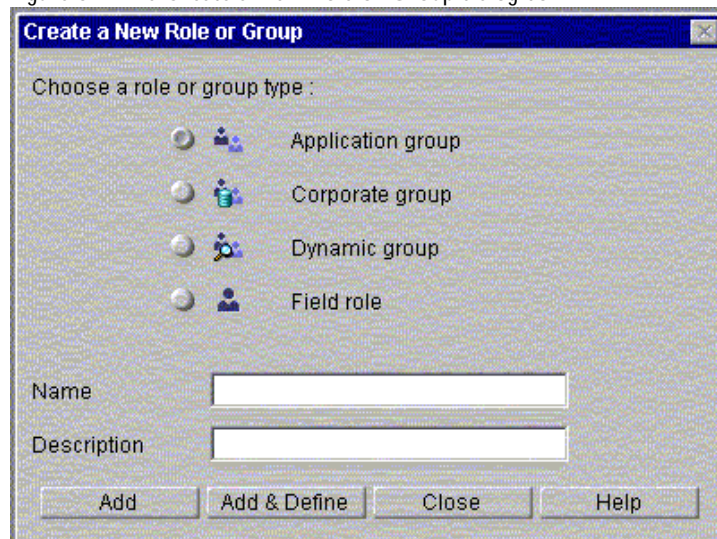
- Click Add to define the item later. The Create a New Role or Group dialog box remains open, and you can add additional items.
4. If you clicked Add & Define, fill in the fields of the next dialog box that appears.

When you close the dialog box, your changes are saved automatically.

The Create a New Role or Group Dialog Box

To supply basic information about the role or group you want to create, enter this information in the “Create a New Role or Group” dialog box, as shown in Figure 5.1.

Figure 5.1 The Create a New Role or Group dialog box



Types of Groups and Roles

There are four different types of groups and roles you can create:

Application group A group defined in a particular application that points to user data in the corporate user directory. The group information is stored in the configuration directory, and the user information is stored in the corporate user directory. Because these groups are stored in the configuration directory, you do not need to have write access to the corporate user directory to create one. The context of this group is the application. Therefore, this group cannot be used by other applications.

Corporate group A group that points to a group defined for the corporation in the corporate user directory. The application's corporate group is exactly the same as the group in the corporate user directory; when the group in the corporate user directory is updated, the corporate group automatically uses those changes. Using corporate groups you can leverage the groups already defined in the corporate user directory. The context of this group is in the corporation; it can be used for many things besides applications.

Dynamic group This group uses a filter to list members in the corporate user directory that match the attributes in the search filter. Unlike a corporate group, which is already defined as a group in the corporate user directory, the dynamic group uses corporate user directory information to create a group dynamically. This group is defined in the context of the corporation (it changes based on changes in the corporate user directory) but it also is used only for the application.

Field role A role represented as a data field. The value of the field varies by process instance, so the role's context is the particular process instance. For example, the creator role (the person who starts a particular process instance) is a field role. The field role can only be one user; you cannot use a field role to assign an activity to a group. Because it's related to a datafield, you cannot create a Field Role if your application has been deployed for testing or production.

Name Enter the name of the role or group. This field cannot contain the following characters: quotation marks ("), commas (,), plus signs (+), and semicolons (;). If you are creating a field role, the name cannot be more than 18 characters long, because the name is also used as the name of the field.

Description An optional longer description of the role or group.

Adding a New Group or Role

To add a new group or role, click either the Add button or the "Add & Define" button.

Use the Add button to add several items in succession, without defining their properties. Each item will be added to the application tree view, but the “Create a New Role or Group” dialog box will remain open so that you can add subsequent items. After adding the desired items, you can define them by double-clicking them in the application tree view.

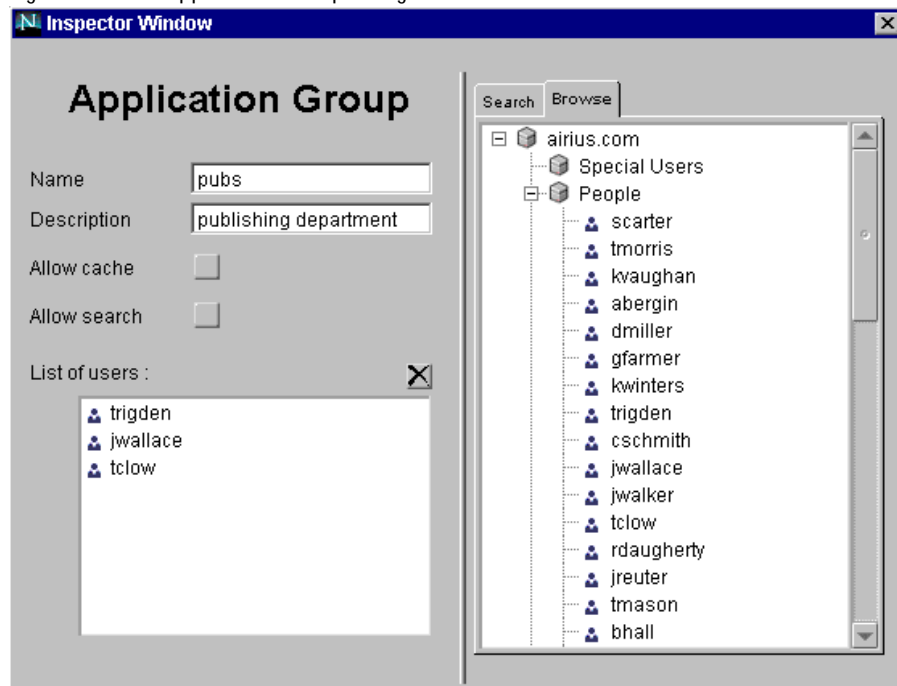
Use the “Add & Define” button to add an item and then immediately define it. Each item is added to the application tree view, but then a new dialog box appears, depending on the kind of group or role you are adding. One of the following dialog boxes will appear:

- The Application Group Dialog Box
- The Corporate Group Dialog Box
- The Dynamic Group Dialog Box
- The Field Role Dialog Box

The Application Group Dialog Box

The Application Group dialog box, shown in Figure 5.2, is where you identify which users are assigned to an application group. For example, if you are designing a time-off request application and have a role for HR approval, you could create an application group with the names of everyone in HR who is allowed to approve a time-off request.

Figure 5.2 The Application Group dialog box with Browse tab



The dialog box contains the following fields, which contain the values you entered in the Create a New Role or Group dialog box:

Name The name of the group.

Description The group's description.

Allow cache If checked, allows group information to be cached. You must also set the User Cache Policy property for the application to All or Members to enable caching. See “Setting Application Properties” on page 65 for more information.

Allow search If checked, allows users in this group to search. If you do not make the group searchable, members of the group will not be able to use the search functionality to check the status of process instances or to find process instances related to specific criteria. They can still use the global search, though. See Chapter 9, “Setting Up Searching,” for more information.

List of users The list of people that are part of this group. This list displays the user RDNs (relative distinguished name) as stored in the corporate user directory. You can add users to the List of Users from the corporate user directory list in the right pane. You can use the Search tab in the right pane to find users, or use the Browse tab to find them.

Note If no users are appearing in your Browse tab or when you search, make sure you have set a corporate user directory in the application's properties.

To see users in the Browse tab, expand the directory and groups by clicking the expansion icons (plus signs). The first time you click the icons the data loads from the corporate user directory. After that, a single click reloads cached data. To reload the information from the corporate user directory, double-click the directory and groups.

If you use the Browse tab, drag the users to the List of Users.

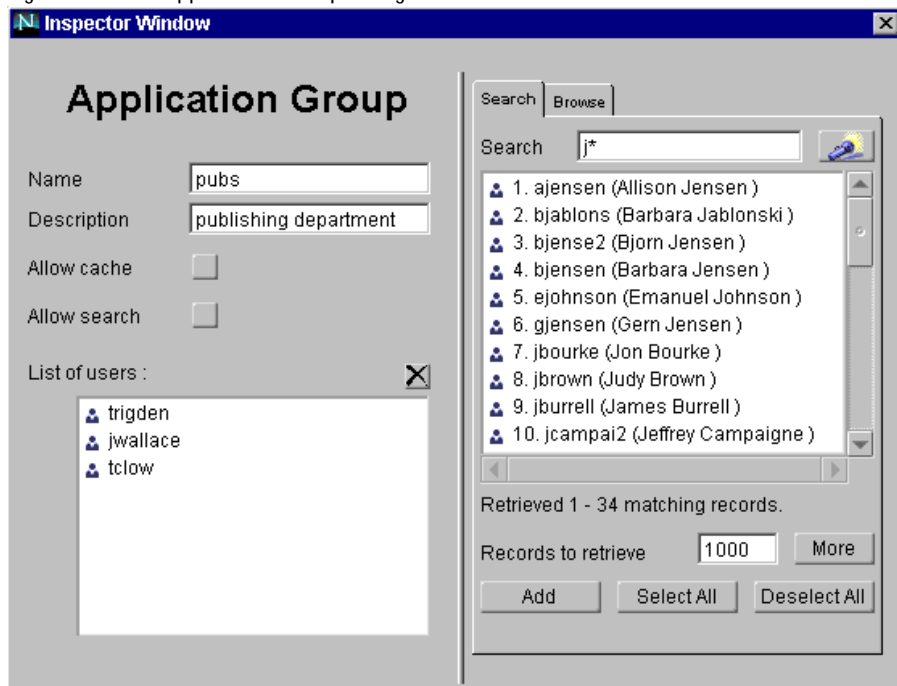
If you use the Search tab, use the buttons at the bottom of the right pane to add users to the List of Users. There are also Select All and Deselect All buttons.

To delete a user from the list of users, select the user name in the List of Users area and click the "X" box above the list.

To use the Search tab to quickly find and add users based on a wildcard pattern, follow these steps:

1. Click the Search tab, which is shown in Figure 5.3.

Figure 5.3 The Application Group dialog box with Search tab



2. Enter a search pattern (such as `j*` or `*jan*`).

This pattern can be for any part of the user's name, such as first name, last name, or user ID.

3. Click the Search icon to get a list of all users that match the pattern.

By default, the list displayed is sorted by user RDN (relative distinguished name). You can specify the sort order in the `preferences.ini` file, which is located in the `builder` folder. Add the following line to the file:

```
sortAttribute = attribute
```

where *attribute* is any LDAP attribute, such as `cn` or `uid`, that you want to appear first in each line of the list. For instance, if you entered `sortAttribute = cn`, the common name appears first in each line displayed.

The search returns the number of records specified in the “Records to retrieve” box. By default, the amount of records retrieved is 1000. You can modify this default in the `preferences.ini` file; add the following line to the file:

```
defaultSearchSize = number
```

where *number* is the amount of records you want displayed by default.

4. Select one or more users from the list.

To select multiple users, highlight a user and without releasing the mouse button drag to the last user you want to add, then release.

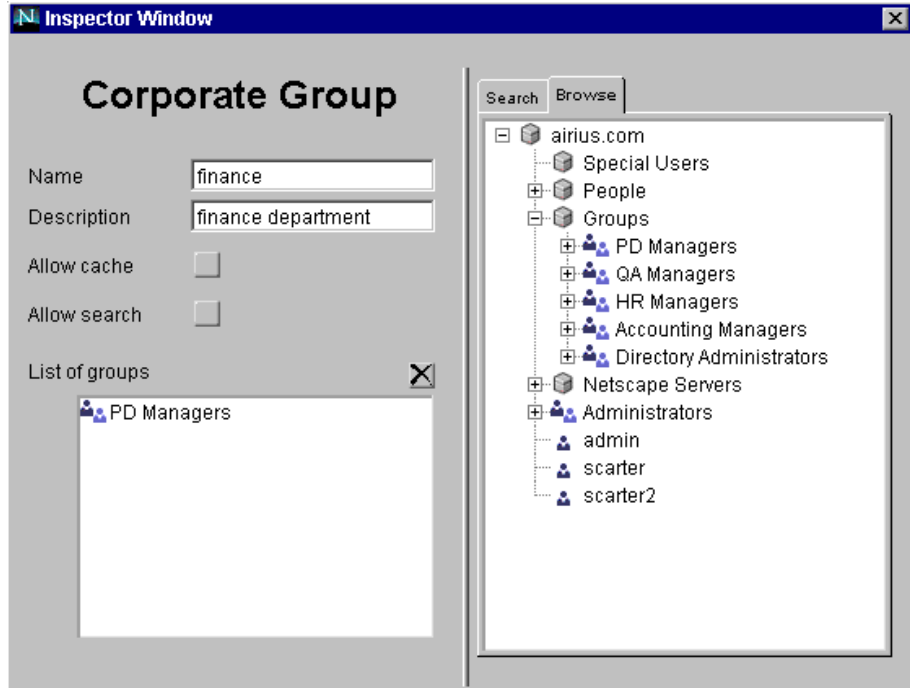
5. Click Add.

The Corporate Group Dialog Box

The Corporate Group dialog box, shown in Figure 5.4, is where you identify a corporate user directory group or set of groups as a group in your application. For example, if you want to have a group called “HR” that consists of all employees in the HR department, and a group like that already exists in the corporate user directory, you can create your group quickly by using the corporate group. Because it is tied to the corporate group, any changes made to the group in the corporate user directory are reflected in the application. Also, since the corporate groups are defined outside of applications, you can use the same group easily for multiple applications.

However, you cannot add or delete users from a corporate group in your application. You must use the corporate group exactly as it is set up in the corporate user directory. To manage your corporate directory, use the administration features of your Directory Server.

Figure 5.4 The Corporate Group dialog box with Browse tab



The dialog box contains the following fields:

Name The name of the group.

Description The group's description.

Allow cache If checked, allows group information to be cached. You must also set the User Cache Policy property for the application to All or Members to enable caching. See “Setting Application Properties” on page 65 for more information.

Allow search If checked, allows users in this group to search. If you do not make the group searchable, members of the group will not be able to use the search functionality to check the status of process instances or to find process instances related to specific criteria. They can still use the global search, though. See Chapter 9, “Setting Up Searching,” for more information.

List of groups The list of groups that are part of this group. You can add a group or groups from the corporate user directory list in the right pane to the List of Groups. You can use the Search tab in the right pane to find groups, or use the Browse tab to find them.

Note If no users are appearing in your Browse tab or when you search, make sure you have set a corporate user directory in the application's properties.

To see users in the Browse tab, expand the directory and groups by clicking the expansion icons (plus signs). The first time you click the icons the data loads from the corporate user directory. After that, a single click reloads cached data. To reload the information from the corporate user directory, double-click the directory and groups.

If you use the Browse tab, drag the groups to the List of Groups.

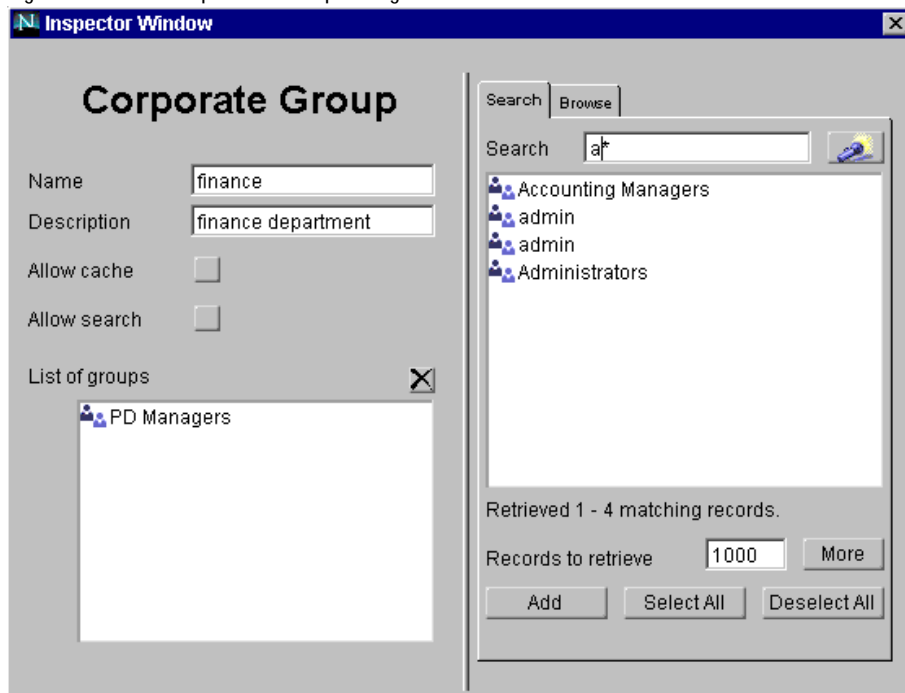
If you use the Search tab, use the buttons at the bottom of the right pane to add groups to the List of groups. There are also Select All and Deselect All buttons.

To delete a group from the List of groups, select the user name and click the "X" box above the List of groups area.

You can use the Search tab to quickly find and add groups based on a wildcard pattern. To do this:

1. Click the Search tab, which is shown in Figure 5.5.

Figure 5.5 The Corporate Group dialog box with Search tab



2. Enter a search pattern (such as `a*` or `*adm*`).
3. Click the Search icon to get a list of all users that match the pattern.

By default, the list displayed is sorted by user RDN (relative distinguished name). You can specify the sort order in the `preferences.ini` file, which is located in the `builder` folder. Add the following line to the file:

```
sortAttribute = attribute
```

where *attribute* is any LDAP attribute, such as `cn` or `uid`, that you want to appear first in each line of the list. For instance, if you entered `sortAttribute = cn`, the common name appears first in each line displayed.

The search returns the number of records specified in the “Records to retrieve” box. By default, the amount of records retrieved is 1000. You can modify this default in the `preferences.ini` file; add the following line to the file:

```
defaultSearchSize = number
```

where *number* is an integer number of records you want displayed by default.

4. Select one or more groups from the list.

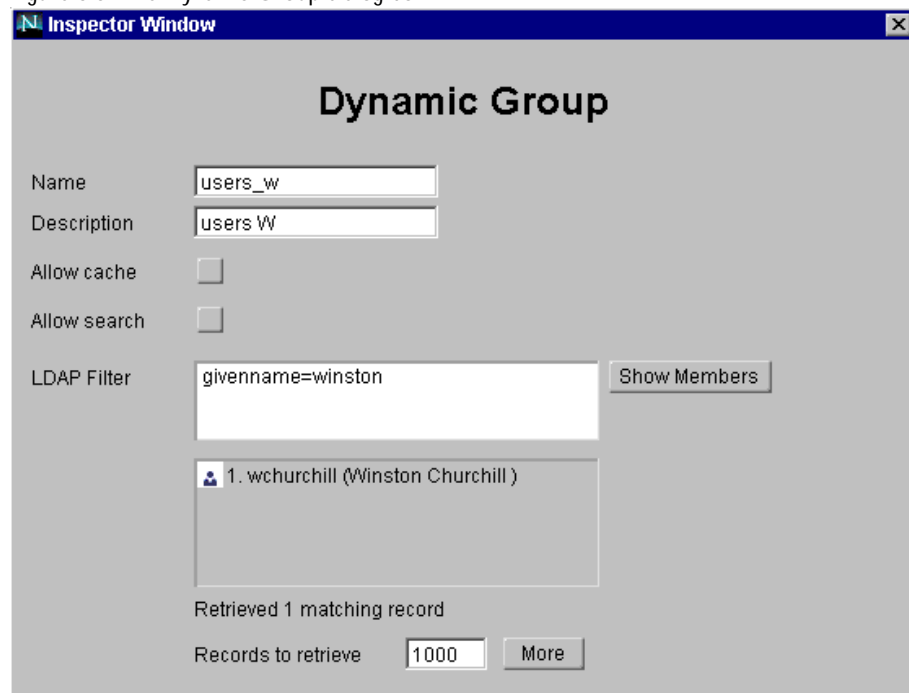
To select multiple users, highlight a user and without releasing the mouse button drag to the last user you want to add, then release.

5. Click Add.

The Dynamic Group Dialog Box

The Dynamic Group dialog box, shown in Figure 5.6, is where you define groups that are created dynamically. The application searches the corporate user directory using an LDAP filter, letting you take advantage of attributes in the corporate user directory.

Figure 5.6 The Dynamic Group dialog box



The dialog box contains the following fields:

Name The name of the group.

Description The group's description.

Allow cache If checked, allows group information to be cached. You must also set the User Cache Policy property for the application to All or Members to enable caching. See "Setting Application Properties" on page 65 for more information.

Allow search If checked, allows users in this group to search. If you do not make the group searchable, members of the group will not be able to use the search functionality to check the status of process instances or to find process instances related to specific criteria. They can still use the global search, though. See Chapter 9, "Setting Up Searching," for more information.

LDAP Filter Enter an LDAP filter and click Show Members to see a list of matches. You do not need to use the Show Members button, but it shows you what your filter finds. Process Builder does not check that the filter you enter is valid. If, after entering the filter string, nothing appears when you click Show Members, either your filter is invalid or it is a valid filter but there are no results that meet the criteria.

A search filter lets you search for an attribute in the corporate user directory. Here are a few sample searches you might use in this field:

- `manager=uid=smith*`
Searches for all users whose manager's user ID is smith*. This syntax only works if the corporate user directory is set up to with the user ID as the RDN's first attribute. If the common name is first, the filter is `manager=cn=smith*`
- `DepartmentNumber=444`
Searches for users whose department number is 444
- `(&(manager=uid=smith*)(employeetype=employee*))`
Searches for people whose manager's user ID is smith* and whose employee type is employee. The ampersand character (&) is the and operator.
- `(|(manager=uid=smith*)(manager=uid=jones*))`

Searches for people whose managers are either smith* or jones*. The pipe (|) character is the or operator.

- `(&(givenname>=j*)(givenname<=o*))`

Searches for people whose names begin with j, k, l, m, n, and o. The less than (<) and greater than or equal to (<=) characters specify the range.

For more information on LDAP filters, see the *Netscape Directory Server Administrator's Guide*.

Records to retrieve The search returns the number of records specified in the “Records to retrieve” box. To obtain the next specified number of records, you click the More button. By default, the amount of records to retrieve is 1000. You can modify this default in the `preferences.ini` file, which is located in the `builder` folder. To modify this value, add this line to the file:

```
defaultSearchSize = number
```

where *number* is the amount of records you want displayed by default.

The Field Role Dialog Box

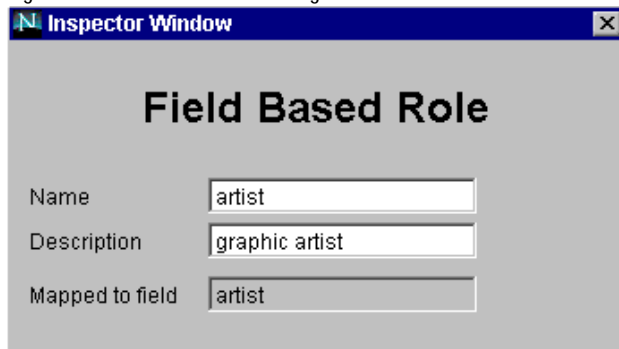
A field role is a role represented as a data field. For a particular process instance, the application uses the value in the data field to determine the user associated with the field role. When you create a field role, Process Builder automatically creates a data field to store the user role.

A field role can be used in an application in either of two ways:

- You can have the end user select a user for the role through the userpicker widget on a form.
- You can set the field role using the functions `setRoleById` or `setRoleByDN`.

Figure 5.7 shows the Field Role dialog box:

Figure 5.7 The Field Role dialog box



The dialog box contains the following fields:

Name The name of the role. This name also becomes the name of the mapped field. The value of this property must be unique among the groups, roles, and data fields that are defined within a given application.

The name can contain only alphanumeric characters with no spaces. It cannot be longer than 18 characters. It's best to use all lower case, and use the underscore character (`_`) as a separator. Also, do not use a name that is a reserved SQL keyword (for example, `select`, `integer`, etc.), or the LiveWire reserved words `project`, `request`, `server`, and `client`.

Description The role's description.

Mapped to field The database field to which you are mapping this role. This field is created automatically, and has the same name as the role. It has the class ID `TextField`.

Prioritizing Groups and Roles

After setting up your groups and roles, you can prioritize them. Usually they are prioritized from most specific to most general. A user can belong to more than one group or role in the same application. Therefore, the priorities determine which forms the user gets.

The priority you set for Groups and Roles determines what order they appear in the Form Access window. See “Setting Access to Forms” on page 172 for more information. To set the order, perform the following steps:

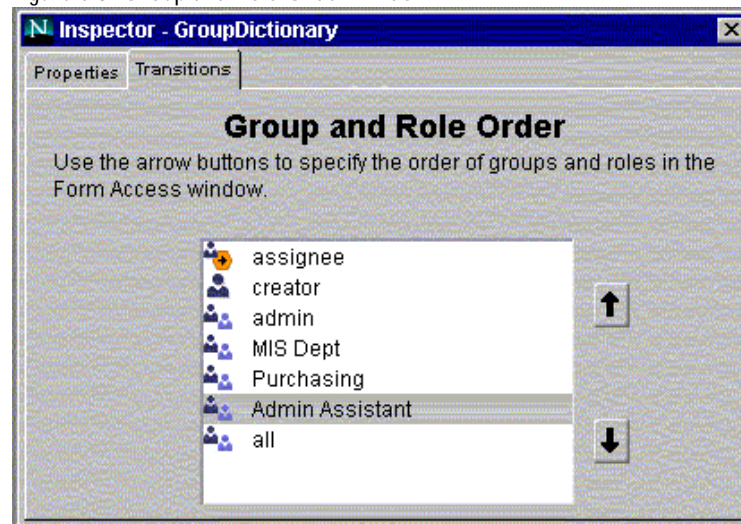
1. In the application tree view, double-click the Groups and Roles folder.

The Inspector window appears.

2. In the Inspector window, click the Transitions tab.

You see all the groups and roles listed, as shown in Figure 5.8:

Figure 5.8 Group and Role Order window



The assignee role is always at the top of the list, and the all role is at the bottom. You cannot change the priorities for these two.

3. To change the priority of a group or role, select it.
4. Click the Up and Down arrows to move the groups and roles. Items at the top of the list have the highest priority.

For example, a user who belongs to both the HR Dept group and the Manager group in the example above gets the form for the HR Dept if it exists. If not, then the user gets the form for the Manager group. If that form doesn't exist, then the user gets the form for "all."

5. Close the window when you're done. Your changes are saved automatically.

The order you set in this window shows up in the Form Access window for forms.

Deleting Groups and Roles

To delete a group or role, follow these steps:

1. In the application tree view, select the group or role.
2. Right-click and choose Delete, or choose Delete from the File menu.

After deleting a field role, you must also delete the corresponding field. However, you cannot delete a field if you have already deployed your application to the Production stage.

Defining Data Fields

This chapter explains how to create data fields for an application and describes the classes of data fields that are available to you.

This chapter includes these sections:

- Data Field Overview
- Creating a Data Field
- Setting Field Properties
- Custom Data Fields with Predefined Class IDs
- Custom Data Fields with Your Own Class ID
- Predefined Data Fields
- Deleting Data Fields
- Setting Up the Content Store

Data Field Overview

The application's data dictionary contains all the data fields that are used by the application and stored in the database. Once you've defined your process map and groups and roles, you start defining the information that is captured and routed throughout the application. If you have a process that currently uses paper forms, a good start is to look at the information gathered on those forms.

A data field can be a structured field in a database, a document, or a file. The data dictionary contains all fields used by an application, regardless of which users see them at which steps in the process.

A field defines how the user interacts with the data (the class), and how the data is stored in the system (the type). Process Builder has some predefined field class IDs, but you can also create your own to extend the field framework.

Note If you are using subprocesses or custom activity you may need to map the data between the parent and child process. See "Subprocesses" on page 86 and "Custom Activities" on page 91 for more information.

Creating a Data Field

To add a field to the data dictionary, you should first ask yourself how the field interacts with the user (which determines the class) and what kind of information the field will store (which determines the type).

To define a field, follow these steps:

1. From the Insert menu, choose Data Field.
2. Fill out the fields in the Create a New Data Field dialog box.

Each data field must have a unique name. The steps are described briefly below; see "The Create a New Data Field Dialog Box" on page 133 for more information.

3. Add a new data field using one of these options:
 - Add a custom data field from predefined classes. Click the radio button Custom Data Field and choose a Class ID from the drop-down list.

- Add a custom data field from a class you create yourself. Click Add New Class, browse to the .zip or .jar file that contains the class you created, and click Open. The class appears in the drop-down list. For information on writing a custom field, see Chapter 18, “Writing Custom Fields.”
 - Add a predefined data field from a template. Click the radio button Predefined Data Field and choose a predefined type from the drop-down list.
4. Choose a class ID or template.
 5. Click either the Add button or the “Add & Define” button.
 - Click Add & Define if you want to define the new item right away. The Inspector window appears for the data field you added.
 - Click Add to define the item later. The Create a New Data Field dialog box remains open, and you can add more items.
 6. If you clicked Add & Define, define the properties in the Inspector window that appears.

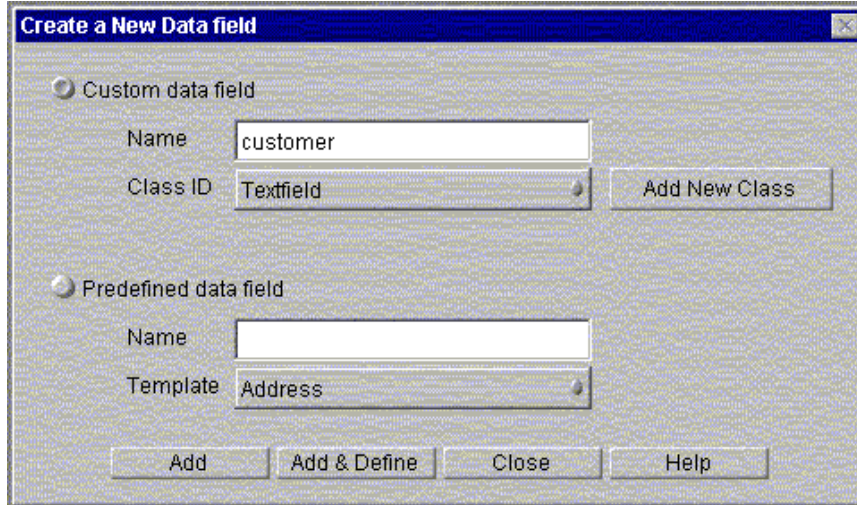
When you close the window, your changes are saved automatically.

The new data field is now listed in the Data Dictionary folder in the application tree view.

The Create a New Data Field Dialog Box

The fields you fill out in this dialog box (see Figure 6.1) depend upon whether you are creating a custom data field or a predefined data field. After you add a field, you can define its properties in the Inspector window.

Figure 6.1 The Create a New Data Field dialog box



Creating a Custom Data Field

To create a custom data field, enter values for the following fields:

Name of this Field The name of the data field. It is also used to name the database column that is going to store this field information. The value of this property should be unique among the data fields that are defined within a given application.

The name can contain only alphanumeric characters with no spaces. It cannot be longer than 18 characters. It's best to use all lower case, and use the underscore character (`_`) as a separator. Also, do not use a name that is a reserved SQL keyword (for example, `select`, `integer`, etc.), or the LiveWire reserved words `project`, `request`, `server`, and `client`, or the word `comment`.

Class ID The class of the field you are creating. Technically, the class ID is the Java component that implements the field's format, but you don't need to understand the Java components unless you want to create your own Class ID. To add your own class IDs, click the `Add New Class` button and add your class to the drop-down list of class IDs. For information on writing a custom field, see Chapter 18, "Writing Custom Fields."

Creating a Predefined Data Field

To create a predefined data field, enter values for the following fields:

Name of this Field defines the name of the data field. It is also used to name the database column that is going to store this field information. The value of this property should be unique among the data fields that are defined within a given application.

The name can contain only alphanumeric characters with no spaces. It cannot be longer than 18 characters. It's best to use all lower case, and use the underscore character (`_`) as a separator. Also, do not use a name that is a reserved SQL keyword (for example, `select`, `integer`, etc.).

Template A predefined field created with default data that you can modify. These templates make defining a field easier if you are inserting a field of a specific type: address, telephone number, or name.

Adding a Data Field

To add a new data field, click either the Add button or the “Add & Define” button.

Use the Add button to add several items in succession, without defining their properties. Each item will be added to the application tree view, but the “Create a New Data Field” dialog box will remain open so that you can add subsequent items. After adding the desired items, you can set their properties by double-clicking them in the application tree view.

Use the “Add & Define” button to add an item and then immediately define it. Each item is added to the application tree view, but the Inspector window also appears.

Setting Field Properties

Data fields are listed in the Data Dictionary folder in the Application Tree View window. To set or modify field properties, follow these steps:

1. In the application tree view, in the Data Dictionary folder, double-click the field name, or highlight it and click the Inspector icon.
2. Update the properties.

Some properties are grayed out and you cannot change them. Other properties contain values in angle brackets (<>). Those values are placeholder values. Replace the brackets and the values inside them with your own data.

3. Close the window. Your changes are saved automatically.

The following properties are available for all fields. In addition, fields have properties determined by their class ID. For more information about those additional properties, see the documentation for those properties.

Data Type (Required) Defines the data type of the field (how it is stored in the database). Valid values depend on the Field Class ID and include TEXT, LONGTEXT, INT, FLOAT, DATE, DATETIME, and FILE. These data types map directly to the SQL data types with the exception of FILE. For more information, see your database documentation.

Display Name The name used in Process Express when end users are allowed to search on a field. Use a name that will be clear and meaningful for the end user.

Field Class ID (read only) This property describes the format of the data field as it appears on the form.

Help Message A help message associated with a data field. When users click the field in a form, this message will appear as text at the bottom of the browser window (below the scroll bar).

Name (read only) The data field's name.

Short Description A description of the field. Currently, this property is unused.

Custom Data Fields with Predefined Class IDs

Use this option to design your own fields using the class IDs that are predefined in Process Builder:

- CheckBox
- Computed
- Date
- DateTime
- Digital Signature
- File Attachment
- Java Applet
- Java Bean
- Password
- Radio Buttons
- Select List
- TextArea
- TextField
- URL
- UserPicker Widget

The following sections describes these class IDs in more detail.

CheckBox

Use this class ID to create fields in the form of checkboxes. In addition to the properties listed in “Setting Field Properties” on page 136, CheckBox fields have the following properties:

Default Value The default value of the CheckBox field. Valid values are checked and unchecked.

Label The label that appears to the right of the checkbox. If you would like to create the label by typing it on the form in the form editor, leave this property blank. However, any label you type into this property is automatically included any time you put the field on a form. If you choose to type the label on the form, you need to type it on each form that uses this field.

On click A script that runs every time the user clicks on a checkbox. The script is client-side Javascript that is associated with the `onClick` event handler of the field.

Computed

This class ID is for fields whose values are computed by the application, not entered by the users. In addition, users cannot view these fields. If you want the field to be viewable, use a text field instead.

In addition to the properties listed in “Setting Field Properties” on page 136, Computed fields have the following properties:

Allow Search Specifies if the field can be used as a search criteria. Valid values are true or false.

Default Value Defines the default value of the field.

Length A required property that defines the length of the field and the length of the database column that will be used to store values of that field. It must be an integer between 0 and 2000. Use a value between 0 and 255 for the TEXT data type, and between 0 and 2000 for the LONGTEXT data type. If you want the field to be searchable, use the TEXT type.

Date

This class ID lets the user enter and validate a date.

In addition to the properties listed in “Setting Field Properties” on page 136, Date fields have the following properties:

Allow Search Specifies if the field can be used as a search criteria. Valid values are true or false

Date Format Represents the formatting of the date. Valid values are DD/MM/YYYY or MM/DD/YYYY.

Default Value Defines the default value of the field.

DateTime

This class ID lets the user enter and validate a date and time.

In addition to the properties listed in “Setting Field Properties” on page 136, DateTime fields have the following properties:

Allow Search Specifies if the field can be used as a search criteria. Valid values are true or false

DateTime Format Represents the formatting of the date and time. Valid values are MM/DD/YYYY HH:MI:SS or YYYY/DD/MM HH:MI:SS. MM is the month, DD is the day of the month, YYYY is the year. HH is the hour, MI is the minutes, and SS is the seconds of the time. Note that the user must put a space between the date and the time.

Default Value Defines the default value of the field.

Digital Signature

The Digital Signature field is used for signing data. If the user has a certificate, the data is encrypted. For example, you would want a Digital Signature field if a Vice President has to approve a price. In this example, the price field is a user input field that is included in the List of Signed fields for the VP Signature field. When the VP updates the signature field, all the fields listed in the List of Signed fields are signed.

A Digital Signature field can be signed only by one person for a process instance. Once someone signs the field, the next time the field appears, it shows the name of the person who signed it.

In addition to the properties listed in “Setting Field Properties” on page 136, Digital Signature fields have the following properties:

List of Signed Fields The list of data fields that require an electronic signature. Separate fields with a semicolon (;) between them. All fields included in this list must be on the same form as the Digital Signature field.

File Attachment

Use the File Attachment class ID when the data field is actually a file you want users to attach. This data field provides menus in Process Express that allow end users to upload, download, and view files in a process. Users can use this field if they are accessing Process Express with Netscape Communicator 4.x, or Microsoft Internet Explorer 4.x or 5.x.

A File Attachment field appears in the HTML form as a signed applet. This applet attaches a file to the form. When the end user decides to attach a file to the form, the user clicks the icon for the applet and the applet uploads the file to a special location on the Enterprise Server. All files for a process instance are grouped in the same folder so you can locate and manage them easily.

You specify this location in the URL property of the Content Store. See “Setting Up the Content Store” on page 156 for more information. You specify what kind of file is uploaded in the data field’s File Name property. For each kind of file you want users to be able to attach, you must create a separate data field.

Usage Tips for File Attachments

If you want to use file attachments with Secure Sockets Layer (SSL), your server must be SSL-enabled. To use SSL, end users must use Netscape Communicator to access Process Express. For information on how to enable SSL on your server, see the *Enterprise Server Administrator’s Guide*.

If you have Netscape Application Server installed on the same system as Process Express, you will not be able to use Netscape Communicator with file attachments. To avoid this situation, you can install NAS and Process Express on different systems, or you can use Microsoft Internet Explorer to access Process Express.

You must have the Web Publisher or Remote File Manipulation turned on in your Enterprise Server in order to use file attachments.

Warning You cannot use the Web Publisher’s access control functions for Enterprise Servers version 3.5 or earlier. As a result, any user who knows a file’s URL can access, read, and modify the content of a file that is stored on Enterprise Server 3.5 or earlier.

File Attachment Properties

In addition to the properties listed in “Setting Field Properties” on page 136, File Attachment fields have the following properties:

Alignment Defines the alignment of the file representation in the HTML form. Valid values are: CENTER, TEXTTOP, ABSCENTER, ABSBOTTOM. These position the file representation in the HTML form as centered, aligned with the top of the tallest text on the line, aligned with the center of the text, and aligned with the bottom of the text. The text may be moved to accommodate the alignment. Only one line of text is aligned with the data field. Subsequent lines of text appear below the data field. If no alignment value is specified, the alignment is handled by the browser.

Background Color The background color for the data field display area. Specify an HTML color code in the format #RRGGBB. For example, for pure blue, specify #0000FF. For more information about specifying colors with this format, see the “Color Units” chapter in the HTML Tag Reference at: <http://developer.netscape.com/docs/manuals/htmlguid/index.htm>.

Border Width The thickness of the border to draw around the data field display area in pixels.

Debug Mode If this is set to true, the data field displays debug information in the browser's Java console.

Display Height The height in pixels of the data field display area. If you change this height, the size of the icon in the data field does not change. The display area gets more or less background space around the icon.

Display Width The width in pixels of the data field display area. If you change this width, the size of the icon in the data field does not change. The display area gets more or less background space around the icon. The optimal width is 100 pixels, but this depends on the size of the font and the value of the File name property.

File Name The file name which is displayed under the data field as a label in the HTML page. The extension of the File Name determines what kind of file can be uploaded via this file attachment data field. For example, if the File Name is attachment.html, then the data field can handle HTML files. If it is attachment.gif, then the data field can handle GIF files.

Font Name The font used for all text in the data field display area and also for the menu associated with the data field in Process Express.

Font Size The font size used for all text in the data field display area and also for the menu associated with the data field in Process Express.

Foreground Color The foreground color for the data field display area. The foreground color is used for text and other foreground elements. Specify an HTML color code in the format #RRGGBB. For example, for pure blue, specify #0000FF. For more information about specifying colors with this format, see the “Color Units” chapter in the HTML Tag Reference at:

<http://developer.netscape.com/docs/manuals/htmlguid/index.htm>

Help URL A URL that points to a help page. If this property is defined, then when the data field is displayed in Process Express it has an additional menu item, User Help, that displays the page located at the Help URL. This URL must be an absolute URL. The intention here is that you can provide helpful information to your end users about the file to attach, such as what it is for or what kind of file it should be.

Icon Height The default height for the icon used in the data field display area. You cannot change this value.

Icon Width The default width for the icon used in the data field display area. You cannot change this value.

MAX URL Length The maximum length of the URL for files that are saved with this data field. It defaults to 100 characters. This URL is computed from the content store URL. Hence, this width should be at least $(\text{content_store_url}) + 32 + \text{length}(\text{File name})$ characters long.

Java Applet

You can use a Java Applet to represent a specific field. Each applet has a matching hidden field automatically added in the HTML form. To pass the values back to PAE, the applet developer must update the form each time the field gets updated. This happens automatically when a user fills out a form, but if you update a Java Applet field without having the user fill out the form, you need to update the form. To do this update, use the provided Java Class, `WFForm`, that has a static `UpdateField` method. You need to package an applet in a `.jar` file. Give the applet and the `.jar` file the same name. Specify this name as the value of the Applet Class ID property.

The properties Field Class ID and Applet Class ID are not the same. The Field Class ID specifies the format of the field (in this case `com.netscape.workflow.field.Applet`). The Applet Class ID specifies which applet is used (for example the applet `UserPicker` which is a directory widget).

In addition to the properties listed in “Setting Field Properties” on page 136, Java Applet fields have the following properties:

Applet Class ID A required field that defines the applet that is going to be used to represent the field in the form. Applets need to be packaged in a `.jar` file. The Applet Class ID matches the name of the `.jar` file.

Additional Parameters This required field defines the additional parameters of this applet. Format additional parameters in the "`<PARAM NAME=... VALUE=...>`" format. They cannot include any semicolons (`:`).

Alignment Defines the alignment of the applet in the HTML form. Valid values are: `CENTER`, `TEXTTOP`, `ABSCENTER`, `ABSBOTTOM`. These position the file representation in the HTML form as centered, aligned with the top of the tallest text on the line, aligned with the center of the text, and aligned with the bottom of the text. If none is specified, the alignment is handled by the browser.

Default Value Defines the default value of the field.

Data Length A required property that defines the length of the field and the length of the database column that will be used to store values of that field. Note that if this length is longer than the Display Size, some characters may not be displayed on the form. The Maximum Length must be an integer between 0 and 2000. Use a value between 0 and 255 for the `TEXT` data type, and between 0 and 2000 for the `LONGTEXT` data type. If you want the field to be searchable, use the `TEXT` type.

Height This required field defines the display height of the applet in the HTML form.

Width This required field defines the display width of the applet in the HTML form.

Java Bean

You can use a Java Bean to represent a specific field. The bean needs to be compliant with the specifications presented in the “Java Objects in Crossware” white paper, available on the Netscape DevEdge site at the following URL:

<http://developer.netscape.com/docs/wpapers/index.html>

In addition to the properties listed in “Setting Field Properties” on page 136, Java Bean fields have the following properties:

Additional Parameters This required field defines the additional parameters of this bean. Format additional parameters in the "<PARAM NAME= . . . VALUE= . . . >" format. They cannot include any semicolons (;).

Alignment Defines the alignment of the bean in the HTML form.

Class ID This required field defines the Java Class that is going to be used to represent the field in the form. This is the full class name of the bean (including package information).

Data Length A required property that defines the length of the field and the length of the database column that will be used to store values of that field. Note that if this length is longer than the Display Size, some characters may not be displayed on the form. The Maximum Length must be an integer between 0 and 2000. Use a value between 0 and 255 for the TEXT data type, and between 0 and 2000 for the LONGTEXT data type. If you want the field to be searchable, use the TEXT type.

Default Value Defines the default value of the field.

Height This required field defines the display height of the bean in the HTML form.

Width This required field defines the display width of the bean in the HTML form.

Password

The Password class ID is used for password fields. When the user enters data into this field, or when it is viewable, the field value is shown as asterisks (*****).

In addition to the properties listed in “Setting Field Properties” on page 136, Password fields have the following properties:

Default Value Defines the default value of the field.

Display Size This required field defines the size of the text field in the HTML form.

Maximum Length A required property that defines the length of the field and the length of the database column that will be used to store values of that field. Note that if this length is longer than the Display Size, some characters may not be displayed on the form. The Maximum Length must be an integer between 0 and 2000. Use a value between 0 and 255 for the TEXT data type, and between 0 and 2000 for the LONGTEXT data type. If you want the field to be searchable, use the TEXT type.

On Blur The script that is run every time the field loses the focus within the HTML form. The script is client-side Javascript that is associated with the onBlur event handler of the field.

On Focus The script that is run every time the field gets the focus within the HTML form. The script is client-side Javascript that is associated with the onFocus event handler of the field.

On Value Change The script that is run every time the value of the field is changed. The script is client-side Javascript that is associated with the onChange event handler of the field.

Radio Buttons

The class ID Radio Buttons produces radio buttons, which appear on the end user forms as the button and the text next to them. This class ID is useful when the list of values is well known. You may also consider using the class ID Select List for this kind of information. Select List produces a drop-down list. See “Select List” on page 146 for more information.

In addition to the properties listed in “Setting Field Properties” on page 136, Radio Button fields have the following properties:

Allow Search Specifies if the field can be used as a search criteria. Valid values are true or false.

Default Value Defines the default value of the field.

Maximum Length A required property that defines the length of the field and the length of the database column that will be used to store values of that field. It must be an integer between 0 and 2000. Use a value between 0 and 255 for the TEXT data type, and between 0 and 2000 for the LONGTEXT data type. If you want the field to be searchable, use the TEXT type.

On Blur The script that is run every time the field loses the focus within the HTML form. The script is client-side Javascript that is associated with the onBlur event handler of the field.

On Focus The script that is run every time the field gets the focus within the HTML form. The script is client-side Javascript that is associated with the onFocus event handler of the field.

On Value Change The script that is run every time the value of the field is changed. The script is client-side Javascript that is associated with the onChange event handler of the field.

Options A required field that represents the radio button options. To create them, type in all available values separated by semicolons (;). For example, to have a Yes and No radio buttons, in the Options text field, type:

```
Yes ; No
```

Do not put spaces before or after the semicolon.

Select List

The Select List class ID produces a drop-down list. It useful when the list of values are well known. Another good class type for this kind of information is Radio Buttons, which produces radio buttons. See “Radio Buttons” on page 145 for more information.

In addition to the properties listed in “Setting Field Properties” on page 136, Select List fields have the following properties:

Allow Search Specifies if the field can be used as a search criteria. Valid values are true or false.

Default Value Defines the default value of the field.

Length A required property that defines the length of the field and the length of the database column that will be used to store values of that field. It must be an integer between 0 and 2000. Use a value between 0 and 255 for the TEXT data type, and between 0 and 2000 for the LONGTEXT data type. If you want the field to be searchable, use the TEXT type.

On Blur The script that is run every time the field loses the focus within the HTML form. The script is client-side Javascript that is associated with the onBlur event handler of the field.

On Focus The script that is run every time the field gets the focus within the HTML form. The script is client-side Javascript that is associated with the onFocus event handler of the field.

On Value Change The script that is run every time the value of the field is changed. The script is client-side Javascript that is associated with the onChange event handler of the field.

Options A required field that represents the values of the options in the drop-down list. To create the options, type in all values separated by a semicolon (;). For example, to have a Yes and No be the options in a drop-down list, in the Options text field type:

```
Yes ; No
```

Do not put spaces before or after the semicolon.

TextArea

Use this class ID to create large areas where the user can enter text.

In addition to the properties listed in “Setting Field Properties” on page 136, text area fields have the following properties:

Default Value Defines the default value of the field.

Number of columns This required field defines the number of columns of this text area. The number of rows multiplied by the number of columns must be between 0 and 2000 (for LONGTEXT) or 0 and 255 (for TEXT).

Number of rows This required field defines the number of rows of this text area. The number of rows multiplied by the number of columns must be between 0 and 2000. The number of rows multiplied by the number of columns must be between 0 and 255 for the TEXT, and between 0 and 2000 for LONGTEXT.

On Blur The script that is run every time the field loses the focus within the HTML form. The script is client-side Javascript that is associated with the onBlur event handler of the field.

On Focus The script that is run every time the field gets the focus within the HTML form. The script is client-side Javascript that is associated with the onFocus event handler of the field.

On Value Change The script that is run every time the value of the field is changed. The script is client-side Javascript that is associated with the onChange event handler of the field.

TextField

Use this class ID to create text fields. Use the Date and DateTime class IDs for dates and times, not TextField. For longer text areas, use the TextArea class ID

In addition to the properties listed in “Setting Field Properties” on page 136, TextFields have the following properties:

Allow Search Specifies if the field can be used as a search criteria. Valid values are true or false

Default Value Defines the default value of the field.

Display Size This required property defines the size of the TextField in the HTML form.

Maximum Length A required property that defines the length of the field and the length of the database column that will be used to store values of that field. Note that if this length is longer than the Display Size, some characters may not be displayed on the form. The Maximum Length must be an integer between 0 and 2000. Use a value between 0 and 255 for the TEXT data type, and between 0 and 2000 for the LONGTEXT data type. If you want the field to be searchable, use the TEXT type.

On Blur The script that is run every time the field loses the focus within the HTML form. The script is client-side Javascript that is associated with the onBlur event handler of the field.

On Focus The script that is run every time the field gets the focus within the HTML form. The script is client-side Javascript that is associated with the onFocus event handler of the field.

On Value Change The script that is run every time the value of the field is changed. The script is client-side Javascript that is associated with the onChange event handler of the field.

URL

A URL field is an input field in edit mode and a link in view mode. It is used for data that corresponds to a URL and is understood by a browser, for example the protocols http, ftp, ldap, etc.

In addition to the properties listed in “Setting Field Properties” on page 136, URL fields have the following properties:

Default Value Defines the default value of the field.

Display Size This required field defines the size of the text field in the HTML form.

Maximum Length A required property that defines the length of the field and the length of the database column that will be used to store values of that field. Note that if this length is longer than the Display Size, some characters may not be displayed on the form. The Maximum Length must be an integer between 0 and 2000. Use a value between 0 and 255 for the TEXT data type, and between 0 and 2000 for the LONGTEXT data type. If you want the field to be searchable, use the TEXT type.

On Blur The script that is run every time the field loses the focus within the HTML form. The script is client-side Javascript that is associated with the onBlur event handler of the field.

On Focus The script that is run every time the field gets the focus within the HTML form. The script is client-side Javascript that is associated with the onFocus event handler of the field.

On Value Change The script that is run every time the value of the field is changed. The script is client-side Javascript that is associated with the onChange event handler of the field.

Target This required field defines the window in which the URL is open in view mode. For example, NewWindow opens the URL in a new window. The Target property takes the same values as the TARGET attribute in HTML.

UserPicker Widget

UserPicker fields appear as an input field and an Address Book button on a form. The input field is where users enter the user information the field calls for. By using the Attribute property, you can specify what the user needs to put in the field (for example, distinguished name, user ID, email address and so on).

If the user already knows the information they want to put in the field, they type it in. If they don't know exactly what to type, they can search the corporate user directory by pressing the Address Book button. This button brings up a screen for searching the corporate user directory. Once the user finds the information they want, it appears in the input field. That information is stored in the database with the type of TEXT.

Note Because the values are automatically stored with a Data Type of TEXT, there is no Data Type property to specify for this class ID.

In addition to the properties listed in "Setting Field Properties" on page 136, UserPicker fields have the following properties:

Allow Search Specifies if the field can be used as a search criteria. Valid values are true or false

Attribute the LDAP attribute you want to search on. Valid values include the distinguished name (dn), user ID (uid), common name (cn), email address (mail), department number, and manager.

Default Value Defines the default value of the field.

Maximum Length A required property that defines the length of the field and the length of the database column that will be used to store values of that field. Note that if this length is longer than the Display Size, some characters may not be displayed on the form. The Maximum Length must be an integer between 0 and 255.

Custom Data Fields with Your Own Class ID

If the standard data field classes provided with PAE do not fit your needs, you can create your own. For example, you can use custom data fields to access external data sources and to generate dynamic content at entry points. To use custom data fields you need to write Java classes that implement the field.

For example, you could write a database-driven Select List field such that all the options available to the end user are extracted from a database using an SQL query. Another example is a field that gathers database information from an external source

For more information about implementing your own custom fields, see Chapter 18, “Writing Custom Fields.”

Predefined Data Fields

The predefined data fields are templates that have been configured with default data that you can modify after you create the field. For example, the Address data field is a TextField with a default size. Select a template data field from the template list box, enter the name of the field, and click Create.

Process Builder contains the following predefined data fields:

- Address
- Name
- Telephone

Address

The predefined data field for addresses has the following predefined property values. You can change all but the field class ID.

Table 6.1 Predefined address properties

Property	Value
Data Type	Text
Display Name	Address
Field Class ID	TextArea
Short Description	Some Address

In addition to the properties listed in “Setting Field Properties” on page 136, Address fields contain the following properties:

Default Value Defines the default value of the field.

Display Size This required field defines the size of the text field in the HTML form.

Help Message A help message associated with a data field. The user sees it as text displayed in the bottom of the browser window (below the scroll bar).

Number of columns This required field defines the number of columns of this text area. The number of rows multiplied by the number of columns must be between 0 and 2000 (for LONGTEXT) or 0-255 (for TEXT).

Number of rows This required field defines the number of rows of this text area. The number of rows multiplied by the number of columns must be between 0 and 2000. The number of rows multiplied by the number of columns must be between 0 and 255 for the TEXT, and between 0 and 2000 for LONGTEXT.

On Blur The script that is run every time the field loses the focus within the HTML form. The script is client-side Javascript that is associated with the onBlur event handler of the field.

On Focus The script that is run every time the field gets the focus within the HTML form. The script is client-side Javascript that is associated with the onFocus event handler of the field.

On Value Change The script that is run every time the value of the field is changed. The script is client-side Javascript that is associated with the onChange event handler of the field.

Name

The predefined data field for names has the following predefined property values. You can change all but the field class ID.

Table 6.2 Predefined name properties

Property	Value
Data Type	Text
Display Name	Name
Field Class ID	TextArea
Short Description	Some

Name fields contain the following properties:

Allow Search Specifies if the field can be used as a search criteria. Valid values are true or false

Default Value Defines the default value of the field.

Display Size This required field defines the size of the text field in the HTML form.

Help Message A help message associated with a data field. The user sees it as text displayed in the bottom of the browser window (below the scroll bar).

Maximum Length A required property that defines the length of the field and the length of the database column that will be used to store values of that field. Note that if this length is longer than the Display Size, some characters may not be displayed on the form. The Maximum Length must be an integer between 0

and 2000. Use a value between 0 and 255 for the TEXT data type, and between 0 and 2000 for the LONGTEXT data type. If you want the field to be searchable, use the TEXT type.

On Blur The script that is run every time the field loses the focus within the HTML form. The script is client-side Javascript that is associated with the onBlur event handler of the field.

On Focus The script that is run every time the field gets the focus within the HTML form. The script is client-side Javascript that is associated with the onFocus event handler of the field.

On Value Change The script that is run every time the value of the field is changed. The script is client-side Javascript that is associated with the onChange event handler of the field.

Telephone

The predefined data field for telephone numbers has the following predefined property values. You can change all but the field class ID.

Table 6.3 Predefined telephone properties

Property	Value
Data Type	Text
Display Name	Telephone
Default Value	(XXX) XXX-XXXX
Field Class ID	TextField

Telephone fields contain the following additional properties:

Allow Search Specifies if the field can be used as a search criteria. Valid values are true or false

Display Size This required field defines the size of the text field in the HTML form.

Help Message A help message associated with a data field. The user sees it as text displayed in the bottom of the browser window (below the scroll bar).

Maximum Length A required property that defines the length of the field and the length of the database column that will be used to store values of that field. Note that if this length is longer than the Display Size, some characters may not be displayed on the form. The Maximum Length must be an integer between 0 and 2000. Use a value between 0 and 255 for the TEXT data type, and between 0 and 2000 for the LONGTEXT data type. If you want the field to be searchable, use the TEXT type.

On Blur The script that is run every time the field loses the focus within the HTML form. The script is client-side Javascript that is associated with the onBlur event handler of the field.

On Focus The script that is run every time the field gets the focus within the HTML form. The script is client-side Javascript that is associated with the onFocus event handler of the field.

On Value Change The script that is run every time the value of the field is changed. The script is client-side Javascript that is associated with the onChange event handler of the field.

Short Description A description of the field.

Deleting Data Fields

To delete a data field, follow these steps:

1. In the application tree view, select the field.
2. Right-click and choose Delete.

If you delete a field that you have already added to forms, you must also edit the forms to remove the field.

If the field is used by a field role, you must first delete the role, then delete the field.

Setting Up the Content Store

The content store is an HTTP URL where your application stores attached documents (documents are attached by File Attachment data fields).

Before deploying your application, if you have File Attachment data fields you need to set properties in your content store.

To set the properties of the content store, follow these steps:

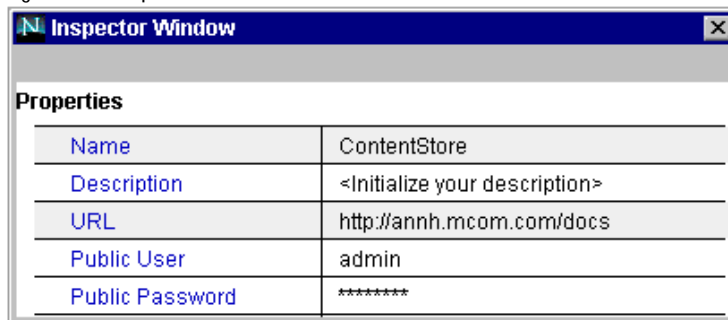
1. In the Application Tree View window, right-click the content store and choose Properties, or double-click content store, or highlight it and click Inspect.
2. Edit the properties.
3. Close the window.

The changes are saved automatically.

The Content Store Inspector Window

Figure 6.2 shows the Inspector window for the content store.

Figure 6.2 Inspector window for the content store



Name The name of the content store. You cannot change this property.

Description The description of the content store. This field does not appear in Process Express or Process Administrator.

URL The URL to the Enterprise Server document directory where you want to put the file attachments, such as `http://server_name/store`. The files for each process instance will be placed in the folder you specify. If the folder does not exist, PAE creates it automatically when the application is deployed. Within the application folder, the documents (numbered by process instances) will be broken into subfolders, each able to contain a maximum of 64000 documents. For example, the first 64000 documents for the application go into a folder named `part001`, the second 64000 documents go into a folder named `part002`, etc.

Public User The Enterprise Server user name used to access the URL where the file attachments are stored. This user must be defined within the Enterprise Server, and must have all access permissions for the directory where the documents are stored.

Public Password The password for the Enterprise Server user defined in the Public User property.

Troubleshooting the Content Store

If users have trouble attaching files in their applications, it could be a problem with access control on the content store. The Enterprise Server location of the content store must have its access permissions set so that the public user can write to it. If you are using a Unix system, you should also check to make sure that the Unix permissions on the folders are set so that the Enterprise Server public user has write access. The Enterprise Server error log may also help you diagnose content store errors.

Warning Because you cannot use the Web Publisher's access control functions for Enterprise Servers version 3.5 or earlier, any user who knows a file's URL can access, read, and modify the content of the file. However, you can use SSL for the content store.

Designing Forms

This chapter explains how to define forms in Process Builder, and how to use the HTML editing features.

This chapter includes these sections:

- Planning Forms
- Creating Forms
- Modifying Forms
- Adding a Banner to Forms
- Setting Access to Forms

Planning Forms

As part of your application, you need to design the forms that the users are going to use to complete work items or monitor the process. You can assign a custom form for each activity or reuse the same form at multiple activities. You can create separate forms for the people participating in the process and the people monitoring the process. The forms define the data the users get to see and how they can interact with the data.

Before you create a form, you need to know the answers to the following questions:

- What data field should be included in the form?
- Should the data be editable, viewable, or hidden? If a field is designated as editable, only the assignee can edit it. It will be viewable to others.
- Is the form for editing data or monitoring the process?
- How should the form look, and where should the data appear in the form?

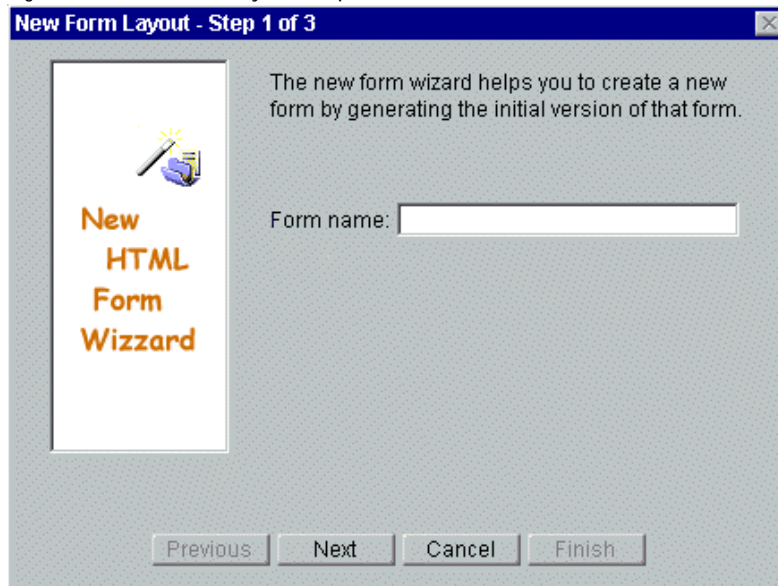
Creating Forms

To create a form, perform the following steps:

1. From the Insert menu, choose Form.

The New HTML Form Wizard appears, as shown in Figure 7.1.

Figure 7.1 New Form Layout, Step 1



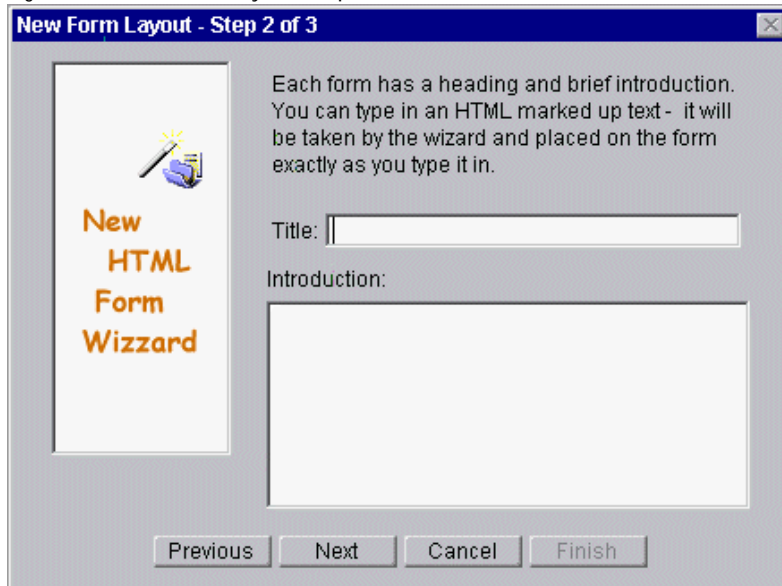
2. Enter a name for your form.

Because the form contains HTML, the name typically ends in `.html`. The name must not exceed 18 alphanumeric characters, including the file extension.

3. Click Next.

Page 2 of the wizard appears, as shown in Figure 7.2.

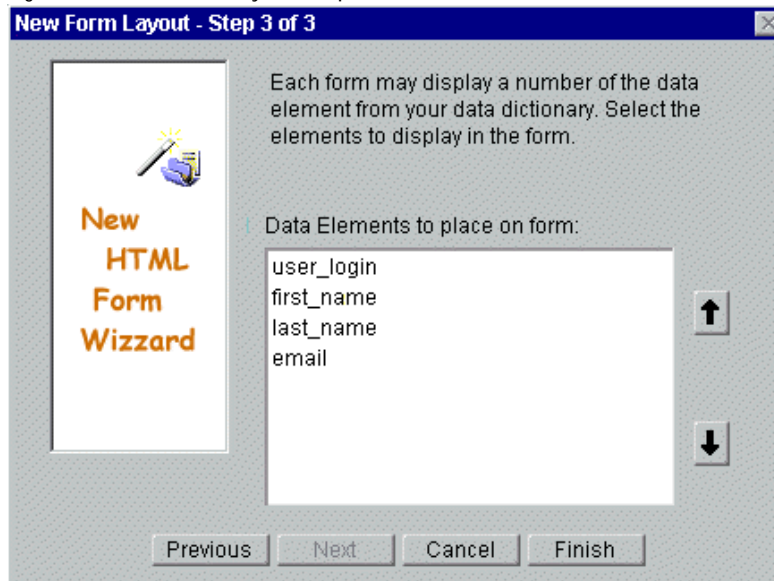
Figure 7.2 New Form Layout, Step 2



4. Enter the title that you want to have appear on the top of the form. Enter some introductory text that explains the form to the users.
5. Click Next.

Page 3 of the wizard appears, as shown in Figure 7.3.

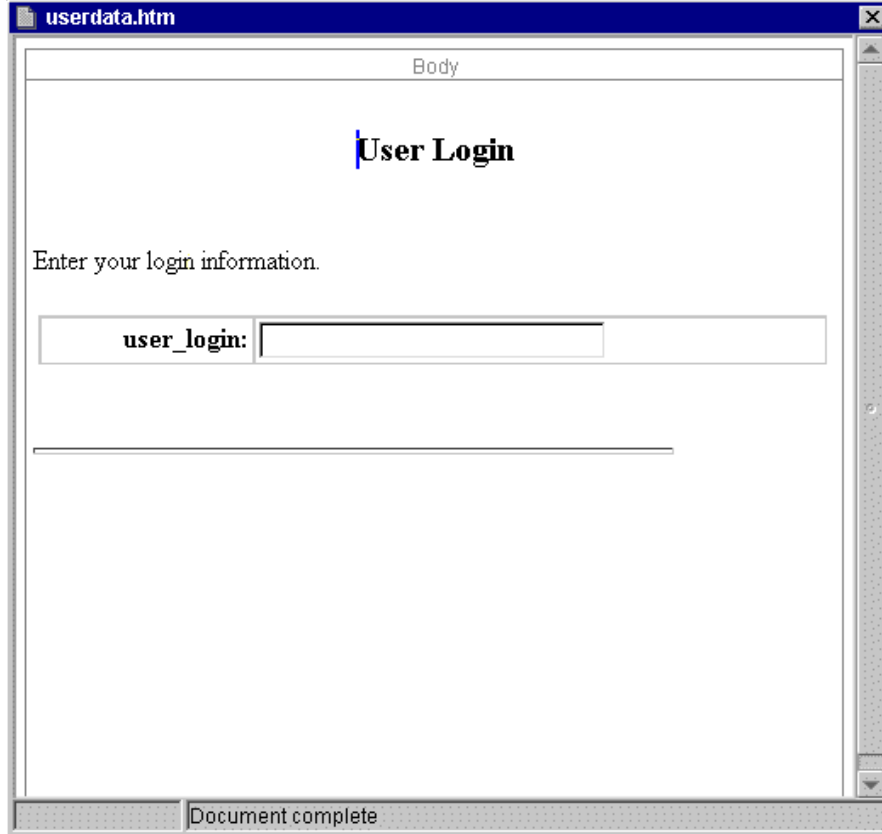
Figure 7.3 New Form Layout, Step 3



- A list of available fields appears, listed in the order they will appear on the form. Highlight the field you want to add to the form.
 - Click the up and down arrow keys to change the placement order of fields on the form.
 - To select multiple data fields, press and hold the mouse, and drag it across the additional field names.
- Click Finish.

The form opens in the form editor for additional editing, as shown in Figure 7.4.

Figure 7.4 Form created by New Form Layout wizard



The screenshot shows a web browser window with the title bar 'userdata.htm'. The main content area is labeled 'Body' and contains the following elements:

- A centered heading: **User Login**
- A prompt: Enter your login information.
- A form field with the label **user_login:** and an adjacent text input box.
- A horizontal line below the input box.

The status bar at the bottom of the window displays 'Document complete'.

8. To make additional changes, edit the form using the HTML editor tools provided in the toolbar.

See “Using the HTML Page Editor Toolbar” on page 166 for an explanation of these tools.

9. Add more data fields by dragging them from the Data Dictionary (in the application tree view) to the form.

10. Set the display mode of the field to edit, view, or hidden.

For more information, see “Changing Field Properties for a Form” on page 169.

11. When you are finished, close the form window and save it.

In addition to the information you create for a form, PAE automatically adds two form elements: a header that describes the work item to the end user, and an action bar at the end of the form. These elements appear when the user sees the form in Process Express. The action bar contains two items: the actions included in the activity, and buttons showing the text of the transitions between actions.

If you have designated the activity as “Allow to Add Comment,” the form also displays a text field for users to add their comments to a process instance. The comments are included in the process instance history.

Modifying Forms

You can modify forms either in Process Builder’s form editing tool or in an external editor.

Using Process Builder’s Form Editor

To modify a form, double-click its name in the application tree view. The form appears in the Page Editor. Each form appears in its own Page Editor window.

You can edit the text on the form using the HTML editor toolbar. You can copy text and fields from one form and paste them in another. You can also use your right mouse button to bring up a menu of editing commands.

Using an External Editor

You can use an external HTML editor to edit the layout of PAE forms for your application with the following cautions:

- You must not change any HTML tags that you or your editor do not understand, and you must leave these tags in all caps: HEAD, BODY, HTML, and SERVER.
- If you use dynamic HTML or JavaScript in a form created with an external HTML editor, you may find that Process Builder corrupts them and you have to use your external editor to clean up the form.

The HTML forms for each application are stored in the folder on your local file system where the application is stored.

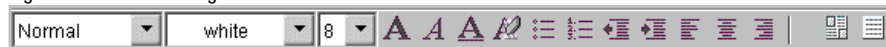
To edit PAE forms using an external HTML editor:

1. Create a form using Process Builder (see “Creating Forms” on page 160) or use a form already created in an external HTML editor.
2. Use Process Builder to add data fields. When you are finished, close the form window and save it.
3. Open the form in your HTML editor.
4. Make and save your changes.

Using the HTML Page Editor Toolbar

The HTML Page Editor toolbar, shown in Figure 7.5, lets you edit and format the HTML forms that you create.

Figure 7.5 HTML Page Editor Toolbar



The drop-down lists and icons assign HTML formatting to the text you choose. You can point your mouse over a toolbar icon to see a “tool tip,” or brief description, of these items. In addition, the drop-down lists and icons are described as follows, from left to right:

- The Style drop-down list lets you format text as normal text, HTML headings, or HTML list items.
- The next two drop-down lists let you choose a color and font size for the text.
- The next three icons (letter A) make text bold, italic, or underlined. The icon following it removes changes to the font size or font style (bold, italic, or underline).
- To the right of the style icons are several formatting icons for creating lists, changing indentation, and changing alignment.

- The last two icons display the form in layout mode (the default) or as HTML source. Layout mode displays the form approximately as you would see it in a browser, with all elements laid out. The source view shows the HTML tags. In this view, you can click a “Reformat Source” button. This makes the HTML code more readable by automatically inserting line breaks as appropriate.

Using the Edit, Insert and Format Menus

The menus at the top of Process Builder also contain commands you can use to modify forms. Since the keyboard shortcuts supported by your operating system may not be available, these menus provide you with editing functionality.

Edit

Use the following Edit Menu items to edit your HTML forms.

Cut Cuts an HTML element from a form.

Copy Copies an HTML element on a form.

Paste Pastes an HTML element into a form.

Delete Deletes the highlighted element or text from the form.

Insert HTML Element

The item on this menu that applies to editing HTML page is HTML Element. You can choose to insert an item from the submenu:

Link Adds an HTML link. To edit the link, use your right mouse button. When you deploy an application, the structure remains the same as in the local Applications folder, so you can use relative links.

Image Adds an image to the HTML page. You can double-click on the image to get a property sheet where you can fill out the image name, source, etc. The image must already be in the application directory, or you can import it using the Import command from the file menu. When you deploy an application, the folder structure remains the same as in the local Applications folder, so you can use relative paths when you import images.

Horizontal Line Adds a horizontal line to the HTML page.

Table (table, row, column, cell) Lets you add a new table (if you are not editing an existing table), or add a row, column, or cell to an existing table.

Client JavaScript Adds client-side JavaScript to the form. Double-click the Script icon in the form, or use your right-mouse button to get to the properties window. In the properties window, click the ellipsis (...) next to the JavaScript Code property to access the editor window. Type your JavaScript there.

Format

These are standard HTML editing commands. Use them to edit forms.

Size Applies the font size you select to the text.

Style Formats text as either bold, italic, underline, or strikethrough.

Remove all styles Removes size and style formatting from text.

Heading Applies an HTML heading tag to the text.

List Applies an HTML list tag to the text. You can choose either a bulleted or a numbered list.

Align Aligns text at the left, right, or center of the page.

Decrease Indent Decreases the amount of indentation.

Increase Indent Increases the amount of indentation.

Using Right-Mouse-Button Menu Commands

When you edit your HTML forms, your right mouse button lets you perform commands.

When you right-click on a data field on a form, you get a pop-up menu with these commands:

Cut Cuts selected object.

Copy Copies selected object.

Paste Pastes previously copied or cut object.

Table Displays extra commands for HTML tables, including commands for cutting, copying, and pasting; for deleting rows, columns and cells; and for setting table, row, and cell properties.

Data Dictionary Displays the data field's inspector window so that you can set any properties. Note that this changes the field's properties on all forms that use it. This option is the same as double-clicking the field in the application tree view.

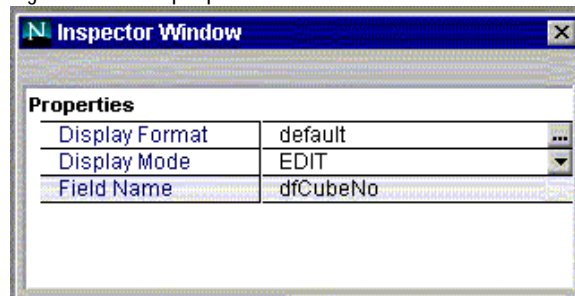
Properties Accesses properties of the field on the form. It allows you to change its display mode: EDIT, VIEW, and HIDDEN. It also shows the Field name and the ID of the component. It lists events and methods associated with the field.

Changing Field Properties for a Form

After adding a field to a form, you can change the field's properties on that form. These changes affect only the field on the form; they do not affect other occurrences of the field within the application.

To display the field properties for that form, double-click the field on the form. The Inspector window appears, as shown in Figure 7.6:

Figure 7.6 Field properties on a form



Display Format The display format of the field. This property is unused for built-in fields. For custom fields that you have implemented through your own Java classes, you can optionally define a display format. Typically, the “default” setting is sufficient.

For information on defining a nondefault display format, see “IPresentationElement Interface” on page 409.

Display Mode The drop-down list contains the following values: VIEW, EDIT, and HIDDEN.

- In view mode, a user can see the field but cannot change it.
- In edit mode, users can edit the field if they are the assignee for that work item. For other users, an edit field appears as a view field.
- In hidden mode, the user cannot see the field. However, the value of the hidden field can be accessed by other fields in the form or by client-side JavaScript scripts. One use for the hidden mode is a signature field, which retrieves the values of the fields that need to be signed and presents these values to the user for signature.

Field Name The name of the field, which you cannot change.

Hints for Setting Field Properties

As you change the display mode, the form updates itself to show the correct visual mode for the field. Editable fields are in boxes on a white background; view-only and hidden fields are on a gray background.

Note that in Process Builder, hidden fields appear on a form. This lets process designers change the mode. It is easier to edit a form if you put all hidden fields at the end of the document. In this way, your view of the form is as close as possible to the user's view.

If you define the same field as editable on more than one form, the value that is stored in the database at the end of the process instance is the last value entered.

Computed fields are always hidden from the user, even in view mode. If you want to display a computed field's information to the user, use a text field instead.

Accessing the Data Dictionary

By right-clicking a field in the form, you get a drop-down list that has a Data Dictionary option. If you choose the Data Dictionary option, you access the properties of the field.

Warning The properties you edit in the data dictionary are shared by all instances of the field. If you change the properties of a field, you change all occurrences of a field on all forms.

Using Scripts to Validate User Input

You can use client-side JavaScript to validate user input on a form as the user submits it. To validate this input, write a client-side JavaScript script called `onSubmitForm()` and insert it into the form. This script is invoked just before the form is submitted.

- If the input is not valid, the script returns `false`, and the form input is not submitted.
- If the input is valid, the script returns `true`, and the form input is submitted.

Adding a Banner to Forms

When an end user sees a form, a banner is included at the top. To create a banner, follow these steps:

1. Create a file to be your banner and name it `banner.gif`.
2. In Process Builder, open the application to which you want to add the banner.
3. From the File menu, choose Import.
4. Click Select and navigate to the `banner.gif` file to import.
5. Enter “images” in the “into folder” field.

6. Click Import.

The banner.gif file is imported into the images folder. If the images folder does not already exist, Process Builder creates it.

Note The file must be named banner.gif and must be in the folder builder/Applications/app_name/images/banner.gif.

Setting Access to Forms

Form Access associates forms with specific users and activities at certain steps in the process. To set access control for your forms, click Form Access, which is located near the bottom of the application tree view. As a result, you see the Form Access window. This window displays form access information in a table, as shown in Figure 7.7.

Figure 7.7 Form Access window

	Create New Data Sheet	Approve Pricing [Product Mgr.]	Approve Pricing [V.P.]	Add Art	Data Sheet Published	Data Sheet Cancelled	Default Exception
assignee	X	approvePricing.html	approvePricingVP.html	addArt.html	X	X	viewWIP.html
creator	X						
admin							
all	newDataSet.html	viewWIP.html	viewWIP.html	viewWIP.html	published.html	cancelled.html	

The row at the top represents steps in the application. The kind of step involved is indicated by an icon. The column on the side represents the groups and roles used by the application. The groups and roles are in the priority order you set when defining groups and roles.

To assign a form to a group or role at a process step, drag a form from the application tree view into the appropriate square in the process map. If you drag a form to the title box of an activity (top row), it shows up in every square in that column that does not already contain a form. If you drag a form to a role/group name box on the left, the form shows up in every square in the row that does not already contain a form. You cannot drag a form to a box that has an X through it.

At each step in the process, the application looks from top to bottom for the appropriate form to present to the user.

For example, in Figure 7.7, in the “Create New Data Sheet” step, the application presents the same form to everyone (all), regardless of their role in the process. Anyone can initiate a datasheet. However, at the “Approve Pricing” step, the assignee (the product manager who needs to approve the pricing) sees a form for approving pricing. Everyone else is only able to see a form that gives read-only information about the work in process.

To remove a form from the table, click the square the form is in and press the delete key.

Forms for Assignees

The forms for the assignees must contain all the information, both editable and view-only, that assignees need in order to complete a step. You must have a special form for the assignee for every activity. Because entry and exit points do not have work items, they do not require an assignee form. These boxes are crossed out, indicating that you cannot add a form to them. For example, in Figure 7.7 you can see that there is a special form for the assignee in all steps except for entry and exit points.

Forms for Monitoring the Process

Often people who are not necessarily participants in the process may want to monitor its progress. In those cases you might create a special form for all interested parties and assign it to the “all” group, or another group that contains people interested in the process.

In other cases, you do not want to give a nonparticipant view because the process involves sensitive information. For example, in an employee time off request application, an employee’s vacation request is private, and should not be available for others to view. In these cases you restrict access by only assigning forms for the people who should have access to the information, for example the assignee and the creator.

Forms for the Administrator

The administrator can perform the following functions on a process instance:

- delegate the work item to a different user
- extend the expiration date of the work item
- change the state of a process instance
- move the work item to a different activity

In order to perform these functions, the administrator must belong to an application’s “admin” group. Usually, you make sure the “admin” group has access to a form at each step of the process, so that the administrators have access to the field information they need to make changes to work items. The form can be either a special form for the “admin” group, or a form for the “all” group.

Setting Access to an Entry Point

For many processes, all employees in a company need to be able to initiate a process instance. For example, every employee needs to be able to submit a time off request. In these cases you set up a form for initiating a process instance and assign it to the “all” group (as shown in Figure 7.7).

In other cases you do not want all users to be able to initiate a process instance. For example, if you had a web site that contained the marketing material for a product suite, you might want only people from the marketing department to be able to add documents to the web site. In that case you might make the form for initiating a process instance available only for a “marketing” group.

Please note that you cannot set a form for the creator at the entry point, since the creator role is not used until after the entry point.

Using Scripts

This chapter describes how to use the scripts available in Process Builder. It also briefly discusses creating your own scripts.

This chapter includes these sections:

- Overview of Scripts
- Predefined Scripts
- Creating Scripts
- Using Client-side Scripts

Overview of Scripts

By using scripts, you can customize and configure your application's activities, automated activities, entry points, exit points, decision points, and transitions. For example, scripts let you assign users to activities and set expiration dates of activities.

Kinds of Scripts

Some scripts are provided with Process Builder, but you can also create your own. All scripts for an application are stored in the Script Dictionary in the application tree view. Table 8.1 describes the available scripts. Each type has its own purpose and its own return value.

Note Each type of script has its own folder within the Script Dictionary. You can drag scripts from one folder to another, but you must update the script so that it returns the data required by the new type before the script will work.

Table 8.1 Available JavaScript scripts

Kind of Script	Purpose	Return Value
Assignment	Assigns a user or users to perform a work item.	Returns an array containing the distinguished names (DNs) of the users to whom this work item is assigned.
Automation	Performs an automated activity. It is executed when the process reaches the automated activity.	Returns <code>true</code> if the activity is successful, otherwise returns <code>false</code> .
Completion	Executes when a work item is completed.	Returns <code>true</code> if the activity is successful, otherwise returns <code>false</code> .
Expiration Setter	Sets the expiration date of a work item.	Returns a Date object.
Expiration Handler	Runs when a task has not been completed before its expiration date is reached.	Returns <code>true</code> if the expiration is successfully handled, otherwise <code>false</code> .
Email Address in Notification	Specifies an email address to which a notification will be sent, either in the To: line or Cc: line.	Returns a valid email address, or a comma-delimited list of valid email addresses. Values can be passed as a string or as a JavaScript array.
Email Subject in Notification	Specifies the subject of an email message.	Returns a string specifying the subject of the email message.
Email Body in Notification	Specifies the body content of an email message.	Returns a string specifying the body of the email message.
Decision point or automation script condition	Specifies a condition that is used to determine which path to follow in the process.	Returns <code>true</code> if the condition is met otherwise <code>false</code>
Initialization script	Specifies tasks to perform when the <i>application</i> first starts.	Returns <code>true</code> if the initialization is successful, otherwise <code>false</code>

Table 8.1 Available JavaScript scripts

Kind of Script	Purpose	Return Value
Shutdown script	Specifies tasks to perform when the <i>application</i> shuts down.	Returns <code>true</code> if the shutdown is successful, otherwise <code>false</code>
Toolkit script	A library function. It can be called by other functions or used as the value of any script for which it returns the appropriate kind of value.	Returns whatever is appropriate.

When to Use Scripts

To edit a script, double-click it in the application tree view, or highlight it and click Inspector. The script text appears in a text edit window. You can also edit scripts using other tools by opening the files where they are stored on your local machine (usually in the `builder/Applications` folder).

Table 8.2 shows what kinds of scripts are available for you to use at each process step.

Table 8.2 Script Types by Process Step

Script Type	Entry Point	User Activity	Automated Activity
Assignment		Available	
Automation			Available
Expiration setter		Available	
Expiration handler		Available	
Completion	Available	Available	Available

About Writing Scripts

You can define your own server-side scripts for any situation where scripts are used. The scripts must be written in JavaScript. You can write them in the script editor in Process Builder. Or you can write them in the editor of your choice, but you must later import them into Process Builder.

Scripts can use any core JavaScript language and objects, such as `array` or `date`. Scripts can also use additional objects that are provided by PAE. These objects include `processInstance`, `workItem`, `corporateDirectory` and `contentStore`. These scripts cannot use client-side objects such as `document` and `window`.

Predefined Scripts

PAE provides a set of predefined server-side scripts, all of which can be accessed from Process Builder. The scripts fall under the following categories:

- Assignment Scripts
- Completion Scripts
- Email Notification Scripts
- Initialization and Shutdown Scripts

Assignment Scripts

Table 8.3 lists the predefined assignment scripts that are available in Process Builder. The documentation below explains what they do, and what parameters you need to supply when you use them to assign a user to an activity. See “Setting Activity Assignments” on page 81 for more information on using assignment scripts in activities. For details on creating assignment scripts, see Appendix A, “JavaScript API Reference.”

Table 8.3 Assignment scripts

Script	Description
<code>randomToGroup</code>	<p>This script selects a user randomly from the users that belong to the group specified by <code>groupname</code>.</p> <p>The parameter to edit is <code>groupname</code>, which is the name of the group that is defined in the Groups and Roles folder. It can be an application group, a corporate group, or a dynamic group. You can improve performance by making the group that is passed as a parameter able to be cached.</p>
<code>toCreator</code>	<p>This script returns a JavaScript array with the Distinguished Name (DN) of the user who created the process instance. <code>toCreator</code> is the default assignment script for a user activity. This script accepts no parameters.</p>
<code>toGroup</code>	<p>This script returns a JavaScript array of Distinguished Names (DNs) of all members of the group specified by the <code>groupName</code> parameter. If this parameter does not correspond to an actual group in the application's Groups and Roles folder, this script returns <code>null</code>.</p>
<code>toManagerOfCreator</code>	<p>This script assigns the work item to the manager of the user who created the process instance. The manager relationship is based on the manager entry of the user in the corporate user directory. For this script to return the manager of the user, the manager attribute of the user in the corporate user directory must contain the distinguished name (DN) of the manager. No parameters need to be edited in this script.</p>
<code>toManagerOfRole</code>	<p>This script assigns the work item to the manager of the person defined in a field role. The manager relationship is based on the manager entry of the user in the corporate user directory. For this script to return the manager of the user, the manager attribute of the user in the corporate user directory must contain the distinguished name (dn) of the manager.</p> <p>The parameter to edit is <code>rolename</code>, which is the name of the field role defined in Groups and Roles folder.</p>

Table 8.3 Assignment scripts

Script	Description
toManagerOf	<p>This script assigns the work item to the manager of the user specified by its user ID. The manager relationship is based on the manager entry of the user in the corporate user directory. For this script to return the manager of the user, the manager attribute of the corporate user directory entry of the user must contain the distinguished name (dn) of the manager.</p> <p>The parameter to edit is <code>userID</code>, which is the user ID of the person whose manager you want to assign an activity to.</p>
toUserById	<p>This script assigns the work item to a user based on the user ID. The parameter to edit is <code>userID</code>, which is the user ID of the person to whom you want to assign the activity.</p>
toUserFromField	<p>This script assigns the work item to the user whose user ID is stored in the <code>datafieldname</code> field of the process instance data. The DN of the user is stored in this field. If you have a field role, using this script has the same affect as assigning the work item to the field role on the Assignment Selection dialog box using the “A group or role” radio button.</p> <p>The parameter to edit is <code>datafieldname</code>, which specifies the name field where the user ID has been stored. This field has to exist in the application data dictionary and has to contain the user ID of the user the work item should be assigned to.</p>
toParallelApproval	<p>This script assigns the activity to all the people specified in <code>arrayOfUserDNs</code>, and tracks if they approve or reject the item they are approving. You also need to use the <code>checkParallelApproval</code> completion script.</p> <p>The parameters to edit are <code>arrayOfUserDNs</code> and <code>FieldName</code>. The <code>arrayOfUserDNs</code> is a JavaScript array which contains the DN's of the people who need to perform the approval. The <code>FieldName</code> is the field that keeps track of who has performed the approval and who still needs to do so. The field is a computed field of length 2000 that you have to add to the data dictionary.</p>

Completion Scripts

There is one predefined completion script, `checkParallelApproval`. It is used for parallel approval.

This script runs when the parallel approval activity is completed. If any user chooses the “stopper action” (that is, refuses to approve the item) the completion script performs the appropriate action. If all users complete the activity without choosing the stopper action (that is, all approve the item) this script performs the appropriate action.

The parameters are `FieldName` and `labelOfStopperAction`. The `FieldName` is the field that keeps track of who has performed the approval and who still needs to do so. The field is a computed field of length 2000 that you have to add to the data dictionary. The `labelOfStopperAction` is the name of the action that a user can take that stops the approval.

For more information on parallel approval, see “Using Parallel Approval” on page 82.

Email Notification Scripts

This section describes the predefined email notification scripts:

- `defaultNotificationHeader()`
- `defaultNotificationSubject()`
- `emailByDN(DN)`
- `emailById(userId)`
- `emailOfAssignees()`
- `emailOfCreator()`
- `emailOfRole(role)`

For examples of using these scripts, see “Assignment, Completion, and Email Scripts” on page 456.

defaultNotificationHeader()

Returns the default notification header for a notification message body. The header contains information about the current work item, such as the current activity name, the process instance ID, the creation date of the process instance and the expiration date (if any).

This function may be used as the notification body script by itself, or may be embedded in your own notification body script. You may also use this function from a template evaluated using `evaluateTemplate()`. The function may only be used successfully from a script associated with a notification; if used anywhere else, an empty string is returned.

The text returned from this function will depend upon the content-type of the notification. If the content-type is `text/html`, the header will be a series of HTML tags; if the content-type is `text/plain`, the header will be plain text.

defaultNotificationSubject()

Returns the default notification subject for the notification subject line. The subject contains information about the current process instance, such as the process instance ID the priority and the title string.

This function may be used as the notification subject script by itself, or may be embedded in your own notification subject script. The function may only be used successfully from a script associated with a notification; if used anywhere else, an empty string is returned.

emailByDN(DN)

Returns a string of comma-delimited email addresses for the user with the given distinguished name (DN). The `mail` attribute for the user must contain a valid email address in the corporate user directory. If the `mail` attribute of the user does not have a value, this function logs an error and returns `null`. This function is intended for use as a notification script, but can be used anywhere that a string of email addresses is needed.

emailById(userId)

Returns a string of comma-delimited email addresses for the user with the given user ID. The `mail` attribute for the user must contain a valid email address in the corporate user directory. If the `mail` attribute of the user does not have a value, this function logs an error and returns `null`. This function is intended for use as a notification script, but can be used anywhere that a string of email addresses is needed.

emailOfAssignees()

Returns a string of comma-delimited email addresses for all the assignees of the work item. The `mail` attribute for each assignee must contain a valid email address in the corporate user directory. If the `mail` attribute is empty for any assignee, no address is added to the string for that assignee. If no assignee has a value in their `mail` attribute, the function logs an error message and returns `null`. This function is intended for use as a notification script, but can be used anywhere that a string of email addresses is needed.

emailOfCreator()

Returns a string of the email address of the user who created the process instance. The user's `mail` attribute must contain a valid email address in the corporate user directory. If the `mail` attribute of the user does not have a value, this function logs an error and returns `null`. This function is intended for use as a notification script, but can be used anywhere that a string of email addresses is needed.

emailOfRole(role)

Returns a string of the email address of the user performing the given role. The user's `mail` attribute must contain a valid email address in the corporate user directory. If the `mail` attribute of the user does not have a value, this function logs an error and returns `null`. This function is intended for use as a notification script, but can be used anywhere that a string of email addresses is needed.

Initialization and Shutdown Scripts

In addition to the predefined assignment scripts, every new application you create has an initialization script and a shutdown script in the script dictionary. The initialization script is called when the application is initialized, and the shutdown script is called when the application stops.

You can use the initialization script to set variables and create objects that are needed for the duration of the application. You use the shutdown script to release any resources allocated by the initialization script, and perform any final cleanup needed.

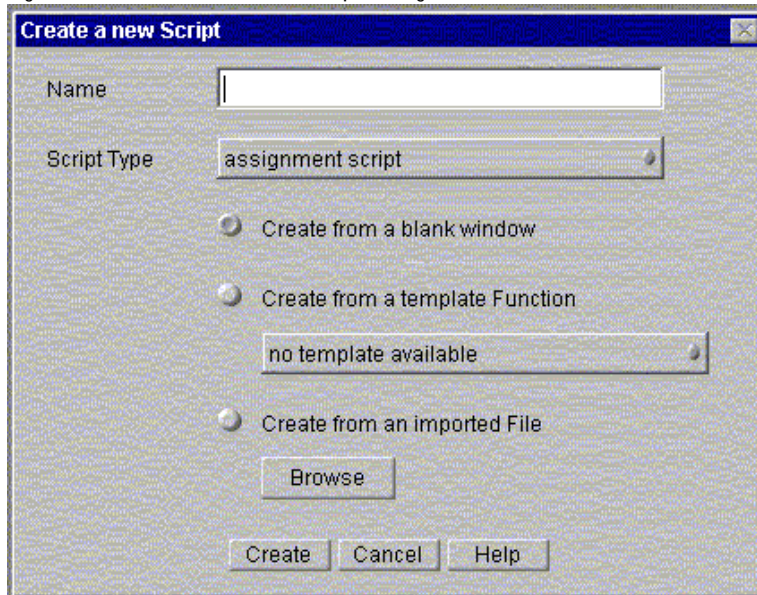
Creating Scripts

In addition to using the predefined scripts described in this chapter, you can also create your own scripts.

To create a script, perform the following steps:

1. From the Insert menu, choose Script. The Create a New Script dialog box appears, as shown in Figure 8.1.
2. Name the script. The name must contain only alphanumeric characters.
3. Choose the type of script you want to create:
 - Assignment Script
 - Automation Script
 - Expiration Setter Script
 - Expiration Handler Script (also known as an Expire script or an On Expire script)
 - Completion Script
 - Toolkit Script

Figure 8.1 The Create a New Script dialog box



4. Click a radio button to choose how you want to create the script.
 - “Create from a blank window” lets you type your script into a blank window, without basing it on an existing script. If you choose this radio button, go to Step 5.
 - “Create from a template function” lets you base your script on another script of the same type which you have previously designated as a template. If you choose this radio button, choose a template from the drop-down list, and go to Step 5.
 - “Create from an imported file” lets you import an existing script from outside the application, or copy one from within the application. If you choose a script from outside the application, you can save the script with the same name. If you choose a script from the current application, you must give the script a unique name. Click Browse to navigate to the .js file you want, and then click Open. Then go to Step 5.
5. Click Create. Your new script appears in the Script Editor Window.

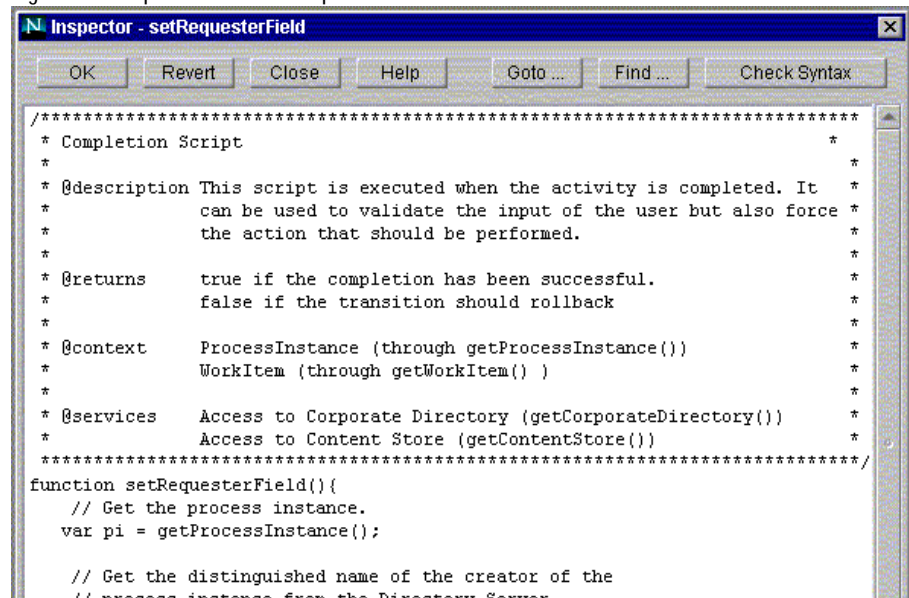
Note The `.js` files you import must follow the conventions for `.js` files in PAE. The name of the file must be the same as the name of the script, and each file can contain only one script.

The Script Editor Window

The script editor window appears when you are creating a new script or editing an existing script. If you are creating a new script from a blank window, the script editor appears with comments in the window pertaining to the type of script you are creating.

The script editor has a text area for typing in your script, and several buttons at the top, as shown in Figure 8.2.

Figure 8.2 A portion of the script editor window



OK Saves changes to the script and closes the window.

Revert Rolls back any changes you've made since the last time you saved the script. The window remains open.

Close Closes the window without saving changes.

Help Launches a web page containing help about this script.

Goto Lets you specify a line number to go to in the script.

Find Lets you specify a text string to search for in the script.

Check Syntax Checks the syntax of the script you've created, and displays an error message if there is an error.

Setting a Script as a Template

If you want to use a script as the basis of future scripts within the current application or other applications, you need to designate it as a template. To set it as a template, follow these steps:

1. In the application tree view, right click on the script name.
2. From the menu, choose "Set as template".

The script is added to the drop-down list of templates that appears when you insert a new script. You can only use a template to create the same type of script. For example, an assignment script template only appears in the drop-down list if you are creating a new assignment script.

If you choose to create a script based on a template, the new script must have a different name from the template script.

When inserting a new script, if you choose to define it from a template function, the body of the template function will be used as the body of the new script. You can then modify the new script to suit your needs.

For example, suppose you define a function called `checkPageCount()` that checks if the value of the `pageCount` field is a number. Then you set it as a template function. Next time you need a script to check that the value of a different field is a number, you could create the new script from the `checkPageCount()` template script, and replace references to `pageCount` by the desired field.

Using Client-side Scripts

Most of this chapter describes server-side scripts that run at various stages in the life cycle of a process instance. However, you may also need to know something about client-side scripts. This section describes information that you may find useful.

Process Builder supports the following client-side scripts:

- Embedded client-side scripts. These are attached by using Process Builder's Insert menu, then the HTML element submenu, and then choosing the Client JavaScript menu command.
- Event handlers for form elements. Event handlers include scripts such as `onValueChange` and `onClick`.

Client-side scripts run in the web page and can access other objects, such as form elements, in the current page. Client-side scripts can use the standard JavaScript language and client-side objects such as `form`, `window` and `document` that are available to all client-side JavaScript scripts.

Every web page displaying a work item contains a single form, which client-side scripts can access using `forms[0]`. For example, the following statement sets the variable `budget` to the value currently showing in the form element named `budget`:

```
var budget = document.forms[0].budget.value;
```

Every data field represented on a form has an associated form element of the same name. For example, if the `AuthorName` field is represented on the form, then the form contains a form element whose name is `AuthorName`. In some cases, this form element is hidden, but in all cases, such a form element exists.

Thus, client-side scripts can access the value of any data field as it is currently displayed in the form. This is done by accessing the value of a form element that has the same name as the data field. Client-side scripts cannot get the value of a data field that is not represented in the form.

For example, the following embedded client-side script displays a message indicating how much money is left on the budget:

```
var budget = document.forms[0].budget.value;  
var amountSpent = document.forms[0].amountSpent.value;  
var amountLeft = budget - amountSpent;
```

```
document.write("<B><FONT COLOR='#CC55BB'> $" + amountLeft + " </FONT></B> ");
```

Figure 8.3 shows the definition of this embedded script in Process Builder, and Figure 8.4 shows the results of this embedded script in Process Express.

Figure 8.3 Defining an embedded client-side script in Process Builder

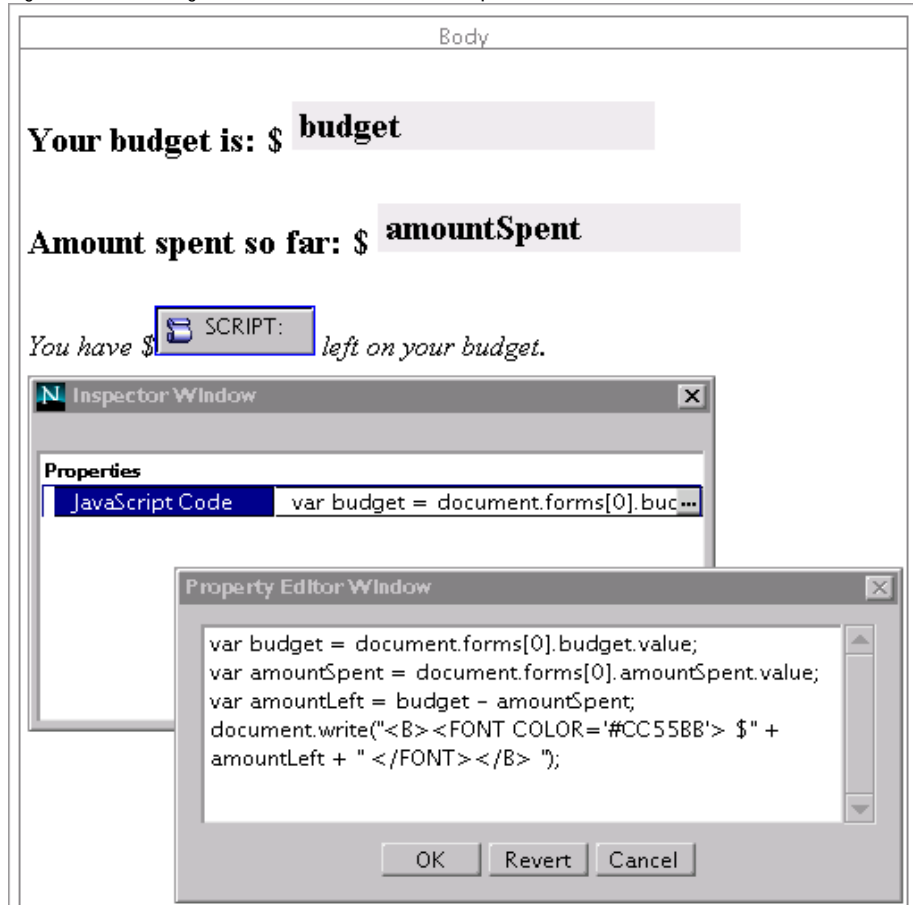


Figure 8.4 Results of an embedded script in Process Express

Your budget is: \$ 5000

Amount spent so far: \$ 1678

*You have \$ **\$3322** left on your budget.*

Note that embedded scripts can access only those form elements that precede it physically in the page, because scripts are executed as the page is laid out. By contrast, event handlers can access other form elements no matter where they are in the page.

Setting Up Searching

If you want end users to be able to search their process instance data, you need to design the application to enable the search capability.

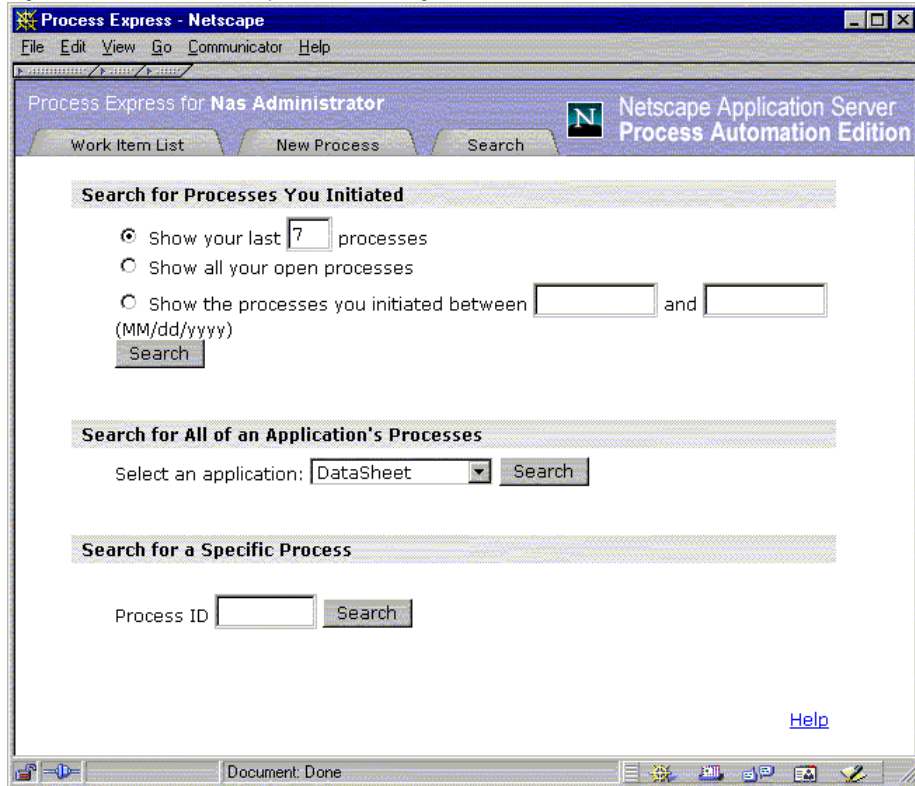
This chapter includes these sections:

- Types of Searching
- Enabling Searching

Types of Searching

You can make applications and data searchable if you want to allow end users to search applications in Process Express. In each application, you can configure two levels of searching: global searching and application-specific searching. The search page is similar to the one shown in Figure 9.1.

Figure 9.1 The Process Express Search Page



In addition, the user can search by their request number, but you do not have to set up anything in the application to allow that kind of search. It is set up automatically for every application.

Global Searching

Global searching is always available to users when they search using the following options:

- the most recent process instances they created
- all the process instances they created that are still open
- all the process instances they created within a date range

Each search option generates a list of process instances that match its search criteria. Each entry in a list contains a link that displays the process instance and another link that displays the history of the process instance.

Application-Specific Searching

You can also configure an application so that participants can search for particular data within it. For example, if you have a data sheet application, you can configure it so that the user can search all instances of the data sheet process by product model number.

The PAE software automatically generates a search form for an application. The search form lets users search for values of fields if you made the fields searchable when you created the field. For example, if your application includes nine data fields, but you only made two searchable, the user can search for the values in those two fields.

The search form generates a list of process instances that match the search criteria. Each entry in the list contains a link to display the process instance and a link to the history of the process instance.

Enabling Searching

To set up searching for an application, you must do the following:

- Set up the appropriate groups to allow members to perform searches.
- Set up forms so that all users for whom you enable searching have forms at all activities.
- Set the appropriate fields to be searchable.

Allowing a Group to Search

To allow members of a group to perform searches, you must set the group to allow searching. If you do not allow searching, members of the group will not be able to use the search functionality to check the status of process instances or to find process instances related to specific criteria. You can allow searching for Application, Corporate, and Dynamic groups. If users are members of more than one group, they can search as long as one of the groups they belong to allows searching.

To allow members of a group to search, follow these steps:

1. In the application tree view, open the Groups and Roles folder.
2. Double-click a group icon, or highlight the group and click Inspect.
3. In the dialog box for the appropriate group, check the Allow Search checkbox.
4. Close the dialog box. Your changes are saved automatically.

Setting Up Forms

To allow users to get information about a specific step in a process instance, they must have a form they can access for that step. For example, if the creator of a process instance searches for information about it, and the process instance

is at a step where there isn't a form the creator can access, he or she cannot get more information about the step beyond its listing in the search results. If the user has a form for that step, the listing has a link to view more information.

One way to make sure that there is always a way for users to see more information on a process instance is to create forms which give a status of the process and set them up for the "all" group. However, for sensitive work items, you might not want to allow all users to see information during the process. In those cases you would set up forms for users who need to have access to the data.

If you want users to be able to see more than the search listing in the search results, follow these steps:

1. In the application tree view, double-click the Form Access icon, or highlight the icon and click Inspect.

The Form Access window appears.

2. Scan the vertical columns to make sure every user you want to view the data has a form they can see for every activity.

Note that you do not need to assign a form specifically for every role. For example, if you have created a form for the "all" role, the creator and other groups and roles will be able to access that form.

3. If you want to add a form to a column, drag the form from the Application Tree View window to the appropriate box in the Form Access window.

If the form you need hasn't been created, you need to create one and come back to this step. You may have to create a new form if you have not created an appropriate form already.

4. Close the Form Access window.

Allowing Searching for Fields

In order for users to be able to use a field as a criterion when they search, you need to make the field searchable. The following predefined field types can be made searchable:

- Computed
- Date
- DateTime
- Radio Button
- Select List
- TextField
- UserPicker Widget

To make a field searchable, perform the following steps:

1. In the application tree view, double-click the field you want to make searchable, or highlight it and click Inspect.
2. In the Inspector window, set the Allow Search property to true.
3. Close the window. Your changes are saved automatically.

Note If a field is searchable, the field length must not exceed 2000 characters.

If you have created your own field class IDs instead of using the predefined ones, you must include the Allow Search property in the JSB file.

Deploying an Application

This chapter describes how to deploy an application. Deployment is specified through the Deployment Dialog Box. After you deploy an application and test it, you'll typically need to change the application and redeploy it.

This chapter describes the following topics:

- Before You Deploy
- Steps for Deploying an Application
- The Deploy Application Dialog Box
- Revising a Deployed Application
- Redeploying an Application

Before You Deploy

Deploying an application makes it available to test and use. However, before you deploy an application, make sure the following tasks have been performed:

- Set up and Configure PAE
- Deploy Subprocesses First
- Save the Process Map, If Desired
- Fix Application Errors

Set up and Configure PAE

Before you can deploy an application, you must have all the pieces of PAE installed and configured, including the corporate user directory, the database, and Process Administrator. In addition, you must have a cluster set up to deploy it to. For more information on creating a cluster, see the *Administrator's Guide*.

You must also have your cluster information included in your `preferences.ini` file. For more information, see “The preferences.ini File” on page 31.

Deploy Subprocesses First

If you are using subprocesses, the child process must be deployed before you use it in a parent process. In this way, all expected components are in place when you deploy the parent process.

Save the Process Map, If Desired

When an administrator is viewing the Work Items List produced by the Work Items Statistics page, it's convenient for the administrator to have an image of the process map. If you want to provide this image, you must save the process

map as a jpeg file before you deploy the application. To save the process map as a jpeg file, open the process map and choose “Save Process Map to JPEG” from the Applications menu.

Fix Application Errors

Once you have built an application and filled in the necessary configuration information, you can begin deploying the application. During the deployment process, your application syntax is checked. If errors are generated, you must correct them before you can deploy the application. If only warnings are generated (and no errors are generated), you are allowed to deploy the application. However, doing so might lead to problems using the application.

Steps for Deploying an Application

To deploy an application, perform the following steps:

1. Click Deploy, or open the Application menu and choose Deploy.

If errors are reported, you must fix them first. If no errors are reported, the Deploy Application dialog box appears. For details about this dialog box, see “The Deploy Application Dialog Box” on page 206.

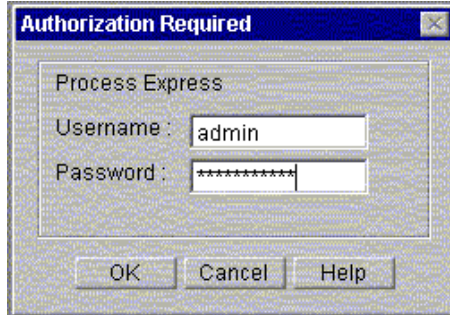
2. From the drop-down menu, select the cluster to which you want to deploy the application.

If this is the first time you are deploying an application, the Authorization Required dialog box appears, prompting you to enter a Process Express username and password.

3. Enter your PAE administration username and password.

As an alternative, you can enter the username and password of someone with deployment privileges. By default, any administrator can deploy an application. A sample entry is shown in Figure 10.1:

Figure 10.1 The Authorization Required dialog box



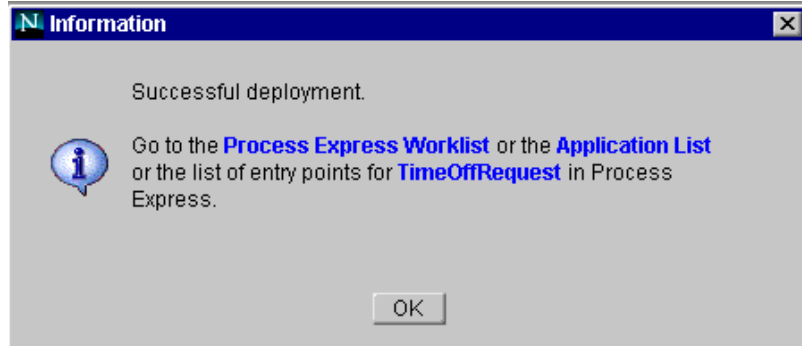
4. Click OK to close the Authorization Required dialog box.
5. In the Deploy Application dialog box, the database user and database password fields are filled in automatically. Change them if necessary.
6. Select a stage for deployment. If you are still designing the application, you typically want to deploy it to Development. If the application is ready for people to use, deploy it to Production.
7. Choose whether the application is in testing mode or not.

If testing mode is set to true, all work items in the application are assigned to the creator of the process instance. If set to false, the work items are assigned to the groups and user roles defined in the application.

8. Click OK.

If deployment succeeds, an Information dialog box appears. A sample is shown in Figure 10.2:

Figure 10.2A successful deployment brings up the Information dialog box



9. From the Information dialog box, you have two main choices:
 - Click OK to close the dialog box and resume working in Process Builder.
 - Click one of the highlighted links (Process Express Worklist, Application List, or the application name) to launch Process Express at a particular location.

Solaris Only

Note: In order for the links to work on Solaris, you must add the following line in your preferences.ini file:

```
browser = netscape_root/netscape -remote openURL{(0)}
```

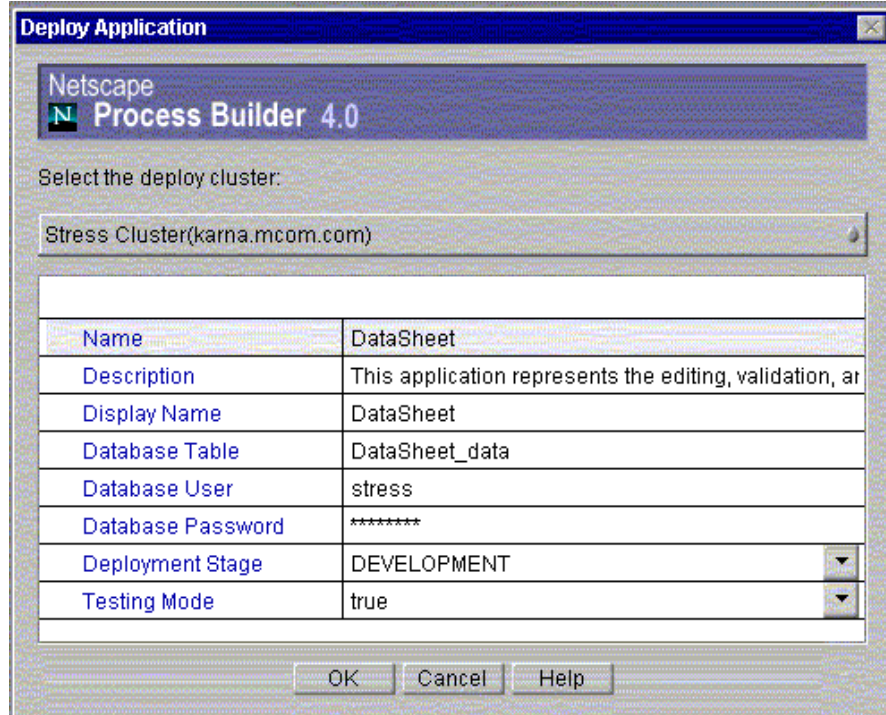
where *netscape_root* is the directory in which your Netscape browser is installed. Without the previous line, your browser will not open when you click the links in the Information dialog box.

After you deploy the application, it is available for use.

The Deploy Application Dialog Box

Figure 10.3 shows the Deploy Application dialog box, where you enter deployment information.

Figure 10.3 The Deploy Application dialog box



Name The name of the application that appears in the list of applications. You cannot edit this field.

Description A longer description of the application. This description appears in Process Express. The user has to have enough information between the Display Name and the Description to identify the application they want from a list of applications, so it's important to make these two fields descriptive.

Display Name The longer, more descriptive name displayed by Process Express. If you are updating an existing application by saving it with a new application name, you can still keep the same Display Name, so that the name users see can be the same from version to version of the application.

Database Table The table in the database where the data for the application is stored. This table is created automatically when you deploy the application. The table name for each application must be unique. This field defaults to the application name. Because the table name can be no more than 15 characters long, if your application name is longer, it truncates the application name. This field is required.

Database User The user name for accessing the database where the application's data is stored. This field is required. This user can be your cluster database user, or you can use a more restrictive application-specific database user. The default is the cluster database user.

Database Password The password used to access the database. This field is required. The default is the cluster database user password.

Deployment Stage Indicates whether the application is in Development or in Production. The deployment stage governs what you can change in the application. In Development, you can change all application information. In Production, there are some application changes you are not allowed to make.

Testing Mode This property designates whether the application is in testing mode or not. If set to true, the application is in testing mode. In testing mode all work items are assigned to the process initiator. You can deploy in testing mode in either the Development or Production stage. This field is required.

Revising a Deployed Application

The deployment stage affects the types of revisions you can make to an application. You can deploy an application either to Development or to Production:

- Development

When you deploy with the deployment stage set to Development, you can test the deployed application. After testing, you can change anything about the application. For example, you can deploy an application, test it, and decide to add or delete data fields. Because the application is still in development, PAE assumes that you do not need to preserve data or open work items. As a result, PAE replaces the whole application when you redeploy it.

For example, suppose you create some process instances using Process Express. Later, if you decide to change a data field in Process Builder and redeploy the application, all existing process instances of that application will be deleted from the database.

- Production

After you deploy an application with a deployment stage of Production, you can still change certain application information if you need to. For example, you can add new activities to an application that is deployed to Production.

Summary of Allowed Revisions

This section lists the different aspects of an application that you can change. The summary lists are grouped as follows:

- Changes to Activities and Transitions
- Changes to Data Elements
- Changes to Forms, Scripts, and Content Store

Changes to Activities and Transitions

Table 10.1 summarizes the revisions allowed to activities and transitions, depending on the application's deployment stage.

Table 10.1 Changes allowed to application's activities and transitions

Task	Development Stage	Production Stage
Add activity	Allowed	Allowed
Add transition	Allowed	Allowed
Remove activity	Allowed	Not allowed
Remove transition	Allowed	Allowed
Rename activity	Allowed	Not allowed
Rename transition	Allowed	Allowed
Remove entry or exit point	Allowed	Allowed

Changes to Data Elements

Table 10.2 summarizes the revisions allowed to data elements, depending on the application's deployment stage.

Table 10.2 Changes allowed to application's data elements

Task	Development Stage	Production Stage
Add data field	Allowed	Allowed
Remove data field	Allowed	Not allowed
Rename data field	Allowed	Not allowed
Change data field type	Allowed	Not allowed
Change data field size	Allowed	Not allowed
Change data field properties	Allowed	Allowed only on the following properties: Allow Search Default Value Display Name Description Help Message On Blur On Focus On Value Change

Changes to Forms, Scripts, and Content Store

Table 10.3 summarizes the changes allowed to forms, scripts, and the content store in deployed applications. These changes are allowed in both the development stage and in the production stage.

Table 10.3 Changes allowed in both development and production stages

Allowed form changes	Allowed script changes	Allowed content store changes
Add form	Add scripts	Change content store URL
Edit form	Edit scripts	Change content store authorization
Delete form	Delete scripts	
Allow user access to form		
Remove user access to form		

Deployed Applications Compared with Local Copies

In most cases, you want to edit the deployed version of an application, not the version stored locally. You'll want to do this because the deployed version is usually the most current. If you edit the local version and then deploy it, you will overwrite your current deployed version.

Note If your local version has a unique application ID that differs from that of your deployed version, you will not be allowed to deploy the local version.

Using a Backup of a Local Application

When you open a deployed application, Process Builder saves a copy of the application locally, in the `builder/Applications/application_name` folder. But you may already have a version of the same application stored locally. In that case, opening the deployed version will launch a dialog box that asks whether you want to overwrite the local version with the deployed version.

If you choose to overwrite the local version, Process Builder saves a copy of the local version in a file called `application_name_backup.zip`. In this way, you'll have a backup of your local version in case you need to revert to it.

For example, the deployed version might be corrupted, or it might not contain recent changes that you made to the local version. To revert to the local backup version, unzip the file.

Saving a Local Application to Another Name

Sometimes you want to make changes to an application, but the changes are not allowed at the deployment stage. In that case, use Save As to rename the application, and then make the changes in the new application.

To rename an application, perform the following steps:

1. Open the most recent version of the application (usually the one deployed to the cluster).
2. From the Application menu, choose Save As.
3. In the dialog box that appears, rename the application.

Note that you can use the same name if you save the application to a different local folder, but the new application will have a different unique ID. You will not be able to overwrite the deployed version of the application with your updated version, even though the name is the same, because the unique ID is different.

4. In the application tree view, double-click the application name to access the properties.
5. Make any changes required to the application, and redeploy under its new name.

Redeploying an Application

After you have deployed an application, you can change it and redeploy it.

- If the deployment stage is Development, redeploying deletes any open work items and data (if they are affected by the application revisions you made).
- If the deployment stage is Production, your open work items and data are preserved.

When you redeploy an application, the application's stage (shown in Process Administrator) is set to the stage of the deployed application, not the locally stored application. For example, if the application in the cluster had a stage of Open, and the local application had a stage of Testing, the redeployed application's stage is set to Open. For more information on application stages, see the *Administrator's Guide*.

The Data Sheet Application

This chapter describes the Data Sheet sample application that is shipped with this product. It includes a general walkthrough of the Data Sheet application, and specific information on the sample application functions.

This chapter includes these sections:

- Data Sheet Application Overview
- Data Sheet Process Map
- Data Sheet Walkthrough
- Groups and Roles
- Data Dictionary
- Forms
- Script Dictionary
- Content Store
- Finished Data Sheet Example
- Configuring the Data Sheet Application

Data Sheet Application Overview

This application automates the process of creating a product data sheet, including the following steps:

- uploading text files and graphics
- formatting the data sheet automatically
- approving the data sheet through two levels of management
- publishing on a web site

The application contains all the necessary forms, user roles, and database fields to complete the process.

Note If you want users to be able to digitally sign the approval form, you must use Netscape Communicator to run this application.

Data Sheet Process Map

The data sheet's process map, shown in Figure 11.1, graphically describes the application's steps.

The process map shows the following steps:

Entry Point The process map contains one entry point for creating a new datasheet.

Activities There are three activities: Add Art, Approve Pricing (Product Mgr) where the Product Manager approves pricing, and Approve Pricing (V.P), where the V.P. approves pricing.

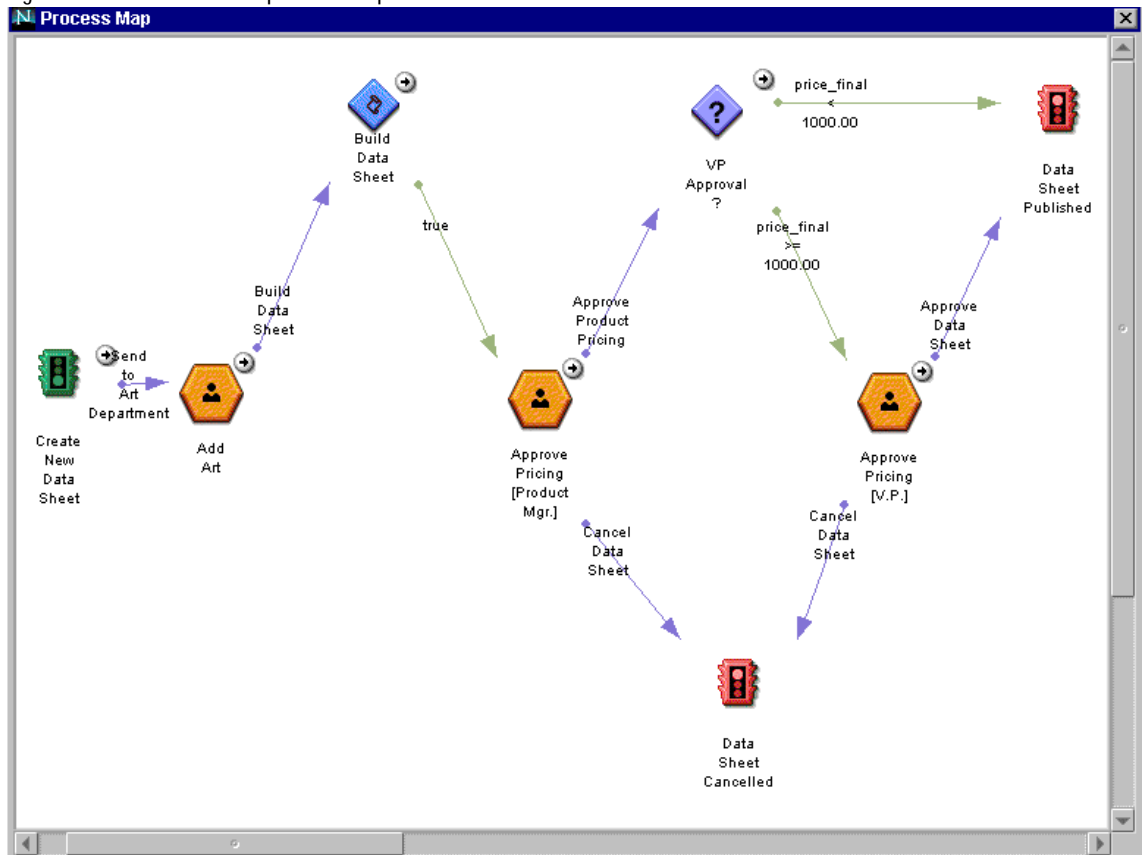
Automated Activity There is one automated activity, called Build Data Sheet, where the data sheet is built.

Decision Point There is one decision point, called VP Approval, which depends upon price. If the price is less than \$1000 dollars, the release is approved. If the price is \$1000 or more, the vice president needs to approve the data sheet before it can be released.

Exit Points There are two exit points in the Data Sheet application, Data Sheet Cancelled that occurs if the V.P. or Product Manager does not approve the data sheet, and Data Sheet Published that occurs if the data sheet passes all approvals and it is published.

Notification There is one notification in the data sheet application. It occurs partway through the process at the product manager approval step.

Figure 11.1 The data sheet process map



Data Sheet Walkthrough

This walkthrough takes you through the process described by the product datasheet application, describing the portions of the application involved in each step.

The Data Sheet Entry Point

Users initiate a data sheet at the Create New Datasheet entry point. When they initiated the data sheet in Process Express, they see the `newDataSheet.html` form. On this form users enter product and pricing information. The product information includes the model number, which is the title field. Process Express displays the title field to identify the process instance.

Note that if you are testing this application yourself, and you sign into Process Express as `admin`, then the creator who initiates the process is set to `admin`.

The following fields are on the `newDataSheet.html` form:

- Product Type (`type`)
- Model Number (`model`)
- Product Description (`description`)
- Base Price (`price`)
- VAR Discount (`discount_code`)

The product description needs to be in the form of an attached text file. The file is stored in the content store and used to create the data sheet.

When users have filled out the form, they click the Send to Art Department button to continue. The button takes its name from the name of the connecting line drawn on the process map between two steps.

The Title Field

The application has a title field, which shows up in Process Express wherever a particular process instance is identified. The process instance title is stored in a special data field, called `title`. It is also set in the application's properties. To see the Title Field in the application's properties, follow these steps:

1. In the application tree view, right click the application name (DataSheet) and choose Properties, or double-click the name, or highlight it and click Inspect.
2. The application's properties appear. The Title Field property set to the "title" data field.

The value of the title field for a process instance is set by the entry point's completion script, `computeTitle`, which puts process instance information into the `title` field. For more information on the script, see "The computeTitle Script" on page 234.

Add Art Activity

The next step is to get the art from the art department. To see the properties for this activity, double-click the activity in the application tree view, or highlight the activity and click Inspect.

Assignment Script

For demonstration purposes, the data sheet activity is assigned to a user with the `toCreator` assignment script. This is one of the standard assignment scripts that ships with Process Builder. It assigns the work item to the creator of the process instance.

Expiration Setter Script

The work item for this step has an expiration date. It uses a hard-coded length of time (1 day) for the expiration date, set by a standard Process Builder script, `expireIn`. It takes the parameters of a number, and a unit of time (days, weeks, etc.). You do not select this script from a drop-down list (as the assignment script is selected). Instead, when you access the wizard for setting expiration dates, you have radio buttons for "Never expire," "Expire in," and

“Defined by a script.” If you choose “Expire in,” there’s a field for a number, and a drop-down list for the unit of time, which you use to set the parameters for the `expireIn` script.

The Forms

Two forms are visible to users at this activity. The assigned user sees the form `addArt.html`. All other users see the `viewWIP.html` (View Work in Progress) form. The various forms are assigned to different groups and users in the Form Access window. See Figure 11.2. You access this window from the application tree view, by double-clicking the Form Access icon.

Figure 11.2 The Data Sheet Form Access window

	Create New Data Sheet	Approve Pricing [Product Mgr.]	Approve Pricing [V.P.]	Add Art	Data Sheet Published	Data Sheet Cancelled	Default Exception
assignee		approvePricing.html	approvePricingVP.html	addArt.html			viewWIP.html
creator							
admin							
all	newDataSheet.html	viewWIP.html	viewWIP.html	viewWIP.html	published.html	cancelled.html	

By reading the grid you can see what role or group sees what form at each step of the application. The groups and roles are in the order specified in the evaluation order in the Groups and Roles properties. For this example, you can see that at the Add Art activity, the assignee sees one form (`addArt.html`), while the “all” group sees another (`ViewWIP.html`).

On the addArt.html form, some fields are display-only, and some the assignee can edit. To see whether or not a field can be edited, select the field on the form and hold down the right mouse button. From there, select the Properties item, and you see a list of the properties of that field in that form. (the Data Dictionary item on this menu allows you to access field properties that are application-wide). The Display Mode property shows whether the field is set to EDIT, VIEW or HIDDEN.

When a field is set to EDIT, only the assignee can actually edit it. If another user views the form, they see the field in VIEW mode.

The only fields that the assignee can edit on this form are layout and art.

The other fields are view-only. The art field is a file attachment field, which means that to complete this field the user attaches a .gif file. The file is stored in the URL specified in the Content Store Properties window. After completing the form, users click the Build Data Sheet button.

On the viewWIP.html form, all fields are view-only. This form is available to “all” so everybody but the assignee sees it. One of the fields, stageURL, gives the URL for the current version of the data sheet, so anyone can monitor its progress.

The Automated Activity

After the user pushes the Build Data Sheet button, the next step is an automated activity called Build Data Sheet. Because this is an automated activity, no user is assigned, and it never becomes a work item on a user’s work list.

To see the properties of the automated activity, in the application tree view, double-click the name of the activity, or highlight it and click Inspect.

When the process reaches the automated activity in the data sheet application, the action takes place immediately (the Schedule and Deferred properties are blank). In addition, the Completion Script property (where another script would be run after the automation script ended) is also blank. In your own applications, those fields can be used to schedule the time the activity takes place, and to perform some additional action when the automated activity is completed.

The Automation Script for this activity is called `buildDS()`. No automation scripts are included in the default script dictionary that is built when you create a new application. The `buildDS` script is unique to the data sheet application.

To look at the script, follow these steps:

1. In the application tree view, open the Script Dictionary.
2. Open the Automated Scripts folder.
3. Double-click `buildDS`, or highlight it and click Inspect. In the Inspector window, you can read and edit the actual text of the script. In addition, you can check the script's syntax to make sure it executes properly.

If the script returns `true` (that is, if it was successfully completed) the process moves on to the next step.

For more information on how the script works, see “The buildDS Script” on page 229.

The Approve Pricing (Product Manager) Activity

In this activity, the product manager approves the pricing on the data sheet.

Assignment Script

For demonstration purposes, the data sheet activity is assigned to a user with the `toCreator` assignment script. This is one of the standard assignment scripts that ships with Process Builder. It assigns the work item to the creator of the process instance. If you are testing this application as the `admin` user, the creator is set to `admin`.

In a non-demonstration situation, the parameter would either be set to the Product Manager's user ID, or a different assignment script could determine the vice president for the product using the corporate user directory and/or the data in the database.

Forms

There are two forms visible to users at this activity. The assignee sees the form `approvePricing.html`. All other users see the `viewWIP.html` form.

On the `approvePricing.html` form, the product manager can see all the pricing data and approve it. The product manager provides authentication by clicking the Sign It button. The field is `signed_by_pm`.

After signing the form, the manager clicks one of two buttons, Approve Product Pricing or Cancel Data Sheet. If the manager chooses Cancel Data Sheet, the data sheet is not published, and it reaches an exit point.

If the manager approves the price, the application moves on to the next step, the decision point.

In this sample, the application does not check to see if the product manager has signed the form. You could build this functionality into your own application by adding a client script to the form called `onSubmitForm()`.

Notification

At the Approve by Process step, a mail notification is sent to the person who started the process instance (the person who filled out the original form at the entry point to create a data sheet). If you are testing this application as the `admin` user, make sure that this user has an email address. For information about providing an email address, see “Adding an Email Attribute for a User” on page 242.

To see the notification’s properties, follow these steps:

1. In the application tree view, open the Process Definition folder.
2. In the Process Definition folder, expand the Approve Pricing (Product Mgr.) activity.
3. You see three items, the two transitions (Cancel Data Sheet and Approve Product Data Sheet) and `NotifyApproving`. Double-click `NotifyApproving`, or highlight it and click Inspect to see the notification’s properties.

In the application tree view, in the Pricing Approval by Process activity, there is a notification called `NotifyApproving`. If you double-click the notification, or highlight it and click Inspect, you see the notification’s properties.

The `Email Address(es)` property instructs the application who to send the notification to. In this case, there is a script called `emailOfCreator()` that mails the message to the person who started the process instance. The message is contained in the `Email Body` property, which in this example contains some sample text.

The VP Approval Decision Point

If the manager approves the data sheet and price, the process takes a different path depending upon the price. If the price is \$1000 or more, the vice president must approve it and it goes to the Approve Pricing (VP) activity.

If the price is less than \$1000, the data sheet is released and published, and reaches an exit point.

The decision point looks at the value in the `price_final` field to determine the price. To see the conditions of the decision point, follow these steps:

1. In the application tree view, in the Process Definition folder, expand the VP Approval ? decision point.
2. You see two transitions, `priceOver1000` and `priceUnder1000`. Right click `priceOver1000` and choose Properties.
3. The transition's properties appear, including the Condition property. The condition property is set to `price_final >= 1000.00`, which means that if the value of the `price_final` field is greater than or equal to 1000.00, this transition is the one that is used.

There is no form, since the decision point takes place automatically.

The Approve Price (VP) Activity

In this activity, the vice president approves the pricing on the data sheet.

Assignment Script

For demonstration purposes, the data sheet activity is assigned to a user with the `toCreator` assignment script. This is one of the standard assignment scripts that ships with Process Builder. It assigns the work item to the creator of the process instance.

In a non-demonstration situation, the parameter would either be set to the vice president's user ID, or a different assignment script could determine the vice president for the product using the corporate user directory and the data in the database.

Forms

There are two forms visible to users at this activity. The assigned user sees the `approvePricingVP.html` form. All other users see the `viewWIP.html` form.

On the `approvePricingVP.html` form, the vice president can see the pricing data and approve it. The vice president provides authentication by clicking the Sign It button. The field is `signed_by_vp`.

Note Digital signing is not supported by Microsoft Internet Explorer.

After signing the form, the vice president clicks one of two buttons, either Approve Data Sheet, or Cancel Data Sheet. If the V.P. selects Cancel Data Sheet, the data sheet does not get published, and it comes to an exit point.

If the manager selects Approve, the data sheet is released and published at the exit point.

Exit Points

This application has two exit points: Data Sheet Cancelled, and Data Sheet Published. If the product manager or vice president does not approve the pricing on the data sheet, the data sheet is canceled and the process ends at Data Sheet Cancelled.

If the product manager and the V.P. (if necessary) approve the pricing, then the data sheet is released and published and the process ends at that exit point.

The exit points each have a form associated with them, `cancelled.html` and `published.html`. These forms allow users to see whether the data sheet was published or not. They also have links to the data sheet on the server, so users can see the final product.

Groups and Roles

The following groups and roles are defined for this application:

- all
- creator
- admin

These are the default groups and roles that are created for every application.

Data Dictionary

The data dictionary contains all the data fields that are used by the application and stored in the database. Table 11.1 lists the data dictionary for the DataSheet sample application. To see the properties for these fields, do the following. In the application tree view, inside the Data Dictionary folder, double-click the name of the field.

Fields are defined to have a class ID, which in turn determines the properties the field has. For more information on the properties for specific field class IDs, see Chapter 6, “Defining Data Fields.”

Table 11.1 Data Sheet Application Fields

Field	Field Class ID
approved_by_pm	computed
approved_by_vp	computed
art	file attachment
discount	textfield
discount_code	select list
discount_desc	textfield
description	file attachment
layout	select list
model	textfield
price	textfield
price_final	textfield
signed_by_pm	signature
signed_by_vp	signature
stageURL	URL
title	textfield
type	radio buttons

When you add fields, you can either add them with predefined class IDs, or you can create your own class IDs. All of the class IDs for fields in the data sheet application are predefined class IDs.

Forms

The Data Sheet application contains the following forms:

- newDataSheet.html
- addArt.html
- approvePricing.html
- viewWIP.html
- approvePricingVP.html
- cancelled.html
- published.html

The walkthrough contained information about what forms users see at what step of the application. To view this information, in the application tree view, double-click Form Access.

To view a form, follow these steps:

1. In the application tree view, double-click the Form Dictionary to open it.
2. In the Form Dictionary, open the form you want by double-clicking it. You see a portion of the form that you can edit.

In addition to the editable part of the form, the user will also see header information and buttons to press.

Figure 11.3 shows the approvePricing.html form as it appears in Process Builder.

Figure 11.3 The approvePricing.html form in Process Builder

approvePricing.html

Body

Product Manager Approval

Product Information

Please review and sign the discount and pricing information related to the product listed here. If the discounted price of this product is higher than \$1,000 then the Vice President of the department will need to approve this pricing.

Product:	Printer AS2015
Complete Datasheet:	D:/SuiteSpot/docs

Product Manager Validation

NOT signed

Base Price:	price
Discount Description :	discount_desc
Discount :	discount
Final Price:	price_final

Document complete

You can see only the body portion of the form. Figure 11.4 shows how the same form appears in Process Express.

Figure 11.4 The approvePricing.html form in Process Express

PM Express for **annh**
Netscape
Process Man

Work Item List
New Process
Search

Process Management
Data Sheet



Product Manager Approval

Product Information

Please review and sign the discount and pricing information related to the product listed here. If the discounted price of this product is higher than \$1,000 then the Vice President of the department will need to approve this pricing.

Product: PrinterAS2015

Complete Datasheet: <D:/SuiteSpot/docs>

Product Manager Validation

NOT
signed

Affix Signature

Base Price: 2000

Discount Description :

Discount :

Final Price:

You can add additional comments here that will appear in the Details & History page:

After completing the above form, select one of the following actions.

Save
Build Data Sheet
Delegate

The form in Process Express has a banner and buttons added to it. In addition, a form may also have a section for comments. The comments are controlled by the Activity's Allow to Add Comments property. If the property is set to true, the comments section appears. Otherwise, it does not appear on the form.

The banner is a file called `banner.gif` found in the application's images folder (`builder/Applications/DataSheet/images`). The banner is created automatically when you create an application, and attached automatically when the form is displayed in Process Express. In your own applications, to change the banner, replace the `banner.gif` file with your own banner file. The file must still be called `banner.gif`.

The buttons are created using the transition names from the process map.

Script Dictionary

The script dictionary contains all the scripts that exist by default in any Process Builder application, and also the additional ones included in the data sheet application.

The data sheet application contains the following scripts in addition to the standard ones:

- `buildDS` (automation script)
- `lookupCode` (toolkit script)
- `buildDataSheet` (toolkit script)
- `computeTitle` (completion script)

These scripts demonstrate how to create scripts of some of the types you will need to create in your own application. They also show how to take advantage of the global scripts, for example, `getProcessInstance()`, which returns a `ProcessInstance` object which can be used to retrieve information about the process instance.

The `buildDS` Script

The automated activity that builds the database uses `buildDS`. If you double-click `buildDS` in the application tree view, you see the text of the script.

```
// Building the datasheet
// 1. LookUp discount information
// 2. Create and publish datasheet
function buildDS()
{
    return lookupCode() && buildDataSheet();
}
```

This script calls other scripts in the toolkit. You add scripts to the toolkit if you want to be able to reuse them within the application.

This script executes each of the following scripts:

- `lookupCode ()` -- sets the final price.
- `buildDataSheet ()` -- builds the datasheet as an HTML file.

The use of the `&&` operator means that the functions are evaluated in order left to right, and so long as each function returns `true`, the next function is evaluated. If all the functions return `true`, the statement returns `true`. If any of the functions return `false`, the remaining functions will not be evaluated, and the function returns `false`.

The lookupCode Script

The `lookupCode` script computes the value of the `price_final` field using the price and discount code that the user entered.

```
function lookupCode () {
    // The global function getProcessInstance()
    // returns the process instance.
    var pi = getProcessInstance();
    // The methods getData and setData of the process instance are used
    // to read the data field and set the value of the datafield. When
    // setting a value, you have to provide the right JavaScript type.
    // Retrieve the discount_code (which is set in the creation form).
    var discount_code = pi.getData("discount_code");
    // Note that discount_code is an element of the data dictionary.
    // Construct the discount description using the discount_code.
    var desc = "Preferred VAR discount (code " + discount_code + ")";
    // The discount_code is something like '10 A'.
    // Extract the numerical discount from the discount_code.
    // Note that parseInt('10 A') returns 10.
```

```

var disc = parseInt(discount_code);
// If the discount is not a number (isNaN)
// add an entry in the error log and return false.
if (isNaN(disc)){
    var e = new Object();
    e.inFunction      = "lookUpCode";
    e.discountCode    = discount_code;
    e.computedDiscount = disc;
    logErrorMsg("ERROR_DISCOUNT_IS_NOT_VALID",e);
    // Return false to ask the engine to rollback the transaction.
    return false;
}
// Update the process instance data by adding the
// discount description, discount rate, and final price.
pi.setData("discount_desc", desc);
pi.setData("discount",disc.toString() );
pi.setData("price_final", Math.round(
    parseFloat(pi.getData("price"))*(100-disc))/100);
// Return true because the script has been successfully performed.
return true ;
}

```

The buildDataSheet Script

This script creates a data sheet as an HTML file. It gets the data from the type, model, price, and discount code values that the user entered in the newDataSheet.html form at the entry point to this process. Additionally, the script opens a file that contains a description of the item in question, and adds the description to the data sheet. The script uses an HTML template file as the template for the datasheet, and ultimately publishes the datasheet as an HTML file.

The HTML template contains place holders identified by \$\$ to indicate places where values of data fields should be plugged into the template. For example, the place holder \$\$model\$\$ will be replaced by the value of the model data field.

For a sample of the description.txt file, see “The description.txt File” on page 235. For a sample of the printer.html template file, see “The printer.html Template File” on page 237

```

function buildDataSheet (){
    // Use the global function getProcessInstance to get a reference to

```

```

// the process instance.
var pi = getProcessInstance();
// Get the value of the description data field, which is a URL
// pointing to where the description text file is located.
var descriptionURL = pi.getData("description");
// Use the global function getContentStore to get a reference
// to the content store.
var cStore = getContentStore();
// Read the content of the description file.
var descriptionContent = cStore.getContent(descriptionURL);
// Get the path name of the application.
var appPath = getApplicationPath();
// Create a connection to the file containing the template to be
// used to format the datasheet. The filename is something like
// appPath/models/printer.html
var template = new File (appPath + "models/" + pi.getData("type")
    + ".html");
// Get the model.
var model = pi.getData("model");
// Construct the URL where the datasheet will be published.
// The global function getBaseForFileName returns the base URL of the
// location in the content store reserved for the process instance.
var dsURL = getBaseForFileName(pi.getInstanceId()) + model
    + ".html";
// Create the string to hold the content for the datasheet.
var dsContent = "<!-- Content of the dataSheet -->";
// Open the template in read-only mode. If the file is not opened
// successfully, log an error and return false.
if (!template.open("r")){
    template.close();
    var e = new Object();
    e.model = model;
    e.template = template;
    e.stagingURL= dsURL;
    logErrorMsg("ERROR_COULD_NOT_OPEN_TEMPLATE_FOR_READ",e);
    return false ;
}
// Merge the template with data field values and with the content in
// the description.txt file. Values of data fields are plugged into
// corresponding placeholders in a template. For example, $$model$$
// in the template is replaced by the value of the model data field.
// Paragraphs in the description.txt file are substituted into

```



```

// the template in place of $$n$$ placeholders. For example, $$2$$ in
// the template is replaced by the second paragraph in
// description.txt.
var partArray = descriptionContent.split("----");
while (!template.eof()){
    // Read a line from the template file and split it into strings
    // using $$ as the separator. Create an array of the strings.
    // param[0] is the first string, param[1] is the second string, etc.
    // For example:
    // <TD WIDTH="30%"> <FONT SIZE=+1>$ $$price_final$$</FONT></TD>
    // param[0] = <TD WIDTH="30%"> <FONT SIZE=+1>$
    // param[1]= price_final
    // param[2] = </FONT></TD>
    var line = template.readLine();
    var param = line.split("$$");
    // If there are 3 sections, substitute either a description
    // paragraph or a datafield value for the middle string.
    if (param.length == 3){
        var ind = parseInt(param[1]); // Returns 0 if not an integer.
        // If ind is > 0 get a paragraph from description.txt
        // else get a data field value.
        var subthis = ind > 0 ? partArray[ind-1] pi.getData(param[1]);
        // Write the first string into the datasheet.
        dsContent += param[0];
        // Write the substituted string into the datasheet.
        dsContent += subthis;
        // Write the last string into the datasheet.
        dsContent += param[2];
    }
    // If there are not 3 strings, it means the line contains no
    // placeholders, so write the whole line into the datasheet.
    else {
        dsContent += line;
    }
    dsContent+="\n";
}
// Close the template.
template.close();
// Use the store method of the contentStore object to publish the
// datasheet to the destination URL that was defined earlier.
var status = cStore.store(dsContent,dsURL);
// Keep track of where the datasheet has been published by storing

```

```

    // the URL in a data field.
    pi.setData("stageURL",dsURL);
    return true;
}

```

The computeTitle Script

This script is invoked every time a data sheet is initiated. It is responsible for initializing the title of the process instance. The title of the process instance shows up in the work list when the application is running in production mode.

This script relies on the fact that the application contains a field called `title` that is used as the title field for the application (that is, in the application's properties, the title property is set to title.)

Users enter the values for the type and model during the entry point. This script runs as a completion script for the entry point. The script simply gets the value of the type and model data fields, and combines them to specify the name of the process instance. For example, if the type is printer and the model is 2345, the title of the process instance is "printer 2345".

```

function computeTitle(){
    // Use the global function getProcessInstance() to get a reference to
    // the process instance.
    var pi = getProcessInstance();
    // Read the value of the field "model" which was set by the
    // participant in the newDataSheet form in the entry point.
    var model = pi.getData("model");
    // Read the value of the field "type" which was set by the
    // participant in the newDataSheet form in the entry point.
    var type = pi.getData("type");
    // Set the value of the field "title" to be an aggregation
    // of the type and the model.
    pi.setData("title", type + " " + model);
    // The execution has been successful.
    return true;
}

```

Content Store

The Content Store has information about where attached files are stored on the Enterprise Server. Before deploying the application, you must fill in the URL, Public User, and Public Password properties. For more information on configuring the Data Sheet sample application for deployment, see “Configuring the Data Sheet Application” on page 241. For more information on content store properties, see “Setting Up the Content Store” on page 156.

Finished Data Sheet Example

The data sheet uses two files that the user adds during the process. These files reside in the `builder/Samples/DataSheet/demo` folder:

- `description.txt`
- `image.gif`

There are also some files shipped with the sample application in the `builder/Samples/DataSheet/models` folder:

- `bar.gif`
- `Printer.html`
- `thumb_printer.gif`

When these files are combined by the application they create a data sheet. The following examples show a sample `description.txt` file, a sample `image.gif` file, and the finished data sheet using these files and the files in the `/models` folder.

The description.txt File

The user uploads a text file as part of completing the form to initiate a new data sheet. Here is an example:

```
Start with superior, high-resolution laser print quality, versatile
paper handling and 12 page-per-minute print speed driven by a fast
processor. What you get is the Aurora LaserJet 2015 printer, <B>Super
```

Fastdelivering outstanding value for individual or shared use from the market leader in laser printing for the office.

The 12 page-per-minute engine and fast processor get performance off to a great start. With the instant-on fuser that eliminates warm-up time, you get a quick 19-second first page out speed. Printing efficiency is further increased with a multi-sheet feeder.

Megabytes of memory effectively doubled with Aurora's Memory Enhancement technology (MEt), and two parallel ports for seamless network compatibility and functionality.

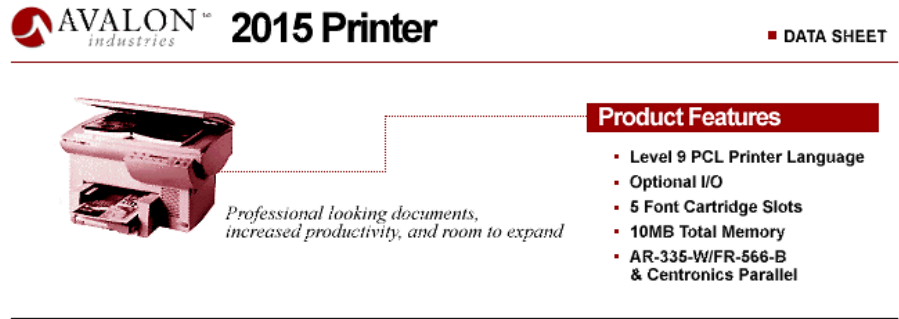
Professional, high-quality printing. Outstanding performance: Compatible, Reliable Aurora printers with the best Image Quality and High Speed printing on the Paper types you want.

<P>True 600 x 600 dpi resolution delivers clear lines, blacker blacks and perfect gradients in all your text and graphics. Brilliant graphics with at least 128 levels of gray for smoother gray transitions and photorealistic images

The image.gif File

The user also adds artwork to the data sheet in the Add Art activity. Figure 11.5 shows an example of the kind of file a user might add:

Figure 11.5 Sample image.gif



This file appears at the top of the finished data sheet.

The printer.html Template File

The printer template file is in the `builder/Samples/DataSheet/models` folder. The data sheet application uses it to format the data sheet. The HTML template file contains placeholders identified by `$$` to indicate places where datafield values or paragraphs from the `description.txt` will appear.

For example, the placeholder `$$model$$` will be replaced by the value of the `model` data field. The place holders indicated as `$$1$$`, `$$2$$`, and so on, indicate places where paragraphs in the `description.txt` file will be plugged into the template. For example, `$$1$$` indicates where the first paragraph of the description will be placed in the data sheet.

The HTML template must have no more than one placeholder per line. Any lines that do not contain placeholders are used in the datasheet without modification.

Finished Data Sheet Example

```
<HTML>
<HEAD>
  <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=iso-
8859-1">
  <META NAME="GENERATOR" CONTENT="Mozilla/4.04[en](X11; U; SunOS 5.5.1
sun4u)[Netscape]">
  <TITLE>$$model$$
    $$type$$ Data Sheet</TITLE>
</HEAD>
<BODY BGCOLOR="white">
<CENTER>
<TABLE BORDER=0 CELLSPACING=0 CELLPADDING=10 WIDTH="650">
<TR ALIGN=LEFT VALIGN=TOP>
<TD COLSPAN=3 VALIGN=CENTER HALGIN=CENTER><IMG SRC="$$art$$"></TD>
</TR>
<TD ALIGN=LEFT VALIGN=TOP>
  $$1$$
</TD>
<TD ALIGN=LEFT VALIGN=TOP>
  $$2$$
</TD>
<TD ALIGN=LEFT VALIGN=TOP>
  $$3$$
</TD> </TR>
</TABLE>

<TABLE BORDER=0 CELLSPACING=0 CELLPADDING=10 WIDTH="650">
<TR><TD COLSPAN=2><IMG SRC="/wfDataSheet_b2/models/bar.gif"></TD></TR>
<TD VALIGN=TOP><IMG SRC="/wfDataSheet_b2/models/thumb_printer.gif"
ALIGN=LEFT></TD>
<TD>
  $$4$$
</TD> </TR>
</TABLE>

<TABLE BORDER=0 CELLPADDING=2 WIDTH="650">
<TR>
  <TD WIDTH="70%" ALIGN=RIGHT><FONT SIZE=+1>Unit Price:</FONT></TD>
  <TD WIDTH="30%"><FONT SIZE=+1> US $ $$price$$ </FONT> </TD>
</TR>
<TR>
  <TD WIDTH="70%" ALIGN=RIGHT><FONT SIZE=+1> VAR Discount:</FONT></TD>
  <TD WIDTH="30%"><FONT SIZE=+1> $$discount_code$$</FONT></TD>
</TR>
</TABLE>
```

```
</TR>
<TR>
  <TD WIDTH="70%" ALIGN=RIGHT><FONT SIZE=+1>Final Price:</FONT></TD>
  <TD WIDTH="30%"> <FONT SIZE=+1>$ $$price_final$$</FONT></TD>
</TR>
<TR><TD COLSPAN=2><IMG SRC="/wfDataSheet_b2/models/bar.gif"></TD></TR>
</TABLE>
</CENTER>
</BODY>
</HTML>
```

The Finished Data Sheet

When the data sheet is built, all the pieces come together. Figure 11.6 shows a sample data sheet which uses the `description.txt` file and the `image.gif` file in earlier examples.

Figure 11.6 The finished data sheet

23456 Printer Data Sheet - Netscape
File Edit View Go Communicator Help

AVALON™ industries 2015 Printer ■ DATA SHEET

Professional looking documents, increased productivity, and room to expand

Product Features

- Level 9 PCL Printer Language
- Optional I/O
- 5 Font Cartridge Slots
- 10MB Total Memory
- AR-335-W/FR-566-B & Centronics Parallel

Start with superior, high-resolution laser print quality, versatile paper handling and 12 page-per-minute print speed driven by a fast processor. What you get is the Aurora LaserJet 2015 printer, **Super Fast** delivering outstanding value for individual or shared use from the market leader in laser printing for the office.

The 12 page-per-minute engine and fast processor get performance off to a great start. With the instant-on fuser that eliminates warm-up time, you get a quick 19-second first page out speed. Printing efficiency is further increased with a multi-sheet feeder.

Megabytes of memory effectively doubled with Aurora's Memory Enhancement technology (MEt), and two parallel ports for seamless network compatibility and functionality.

Professional, high-quality printing. Outstanding performance: Compatible, Reliable Aurora printers with the best Image Quality and High Speed printing on the Paper types you want.

True 600 x 600 dpi resolution delivers clear lines, blacker blacks and perfect gradients in all your text and graphics. Brilliant graphics with at least 128 levels of gray for smoother gray transitions and photorealistic images

Unit Price: US \$ 2000
VAR Discount: 14 B
Final Price: \$ 1720

Document Done

The pricing information comes from information entered by users into forms and approved by the product manager and the vice president, if required.

Configuring the Data Sheet Application

Before you can deploy the Data Sheet application on your own system, you must configure the application. The main steps are as follows:

1. Set the corporate directory as described in “Setting Your Corporate Directory” on page 70.
2. Make sure that your `admin` user has an email address. For details on creating an email address for the `admin` user, see “Adding an Email Attribute for a User” on page 242.
3. Configure the content store. For details, see “Using File Attachments and Content Stores” on page 243.

For demonstration purposes, all the activities are assigned to the creator of the process instance. As a result, if you deploy this application as currently set up, you can create a process instance and participate in all the steps.

Configuration Hints

This section describes the following configuration hints:

- How Users Access the Data Sheet
- Process Express and Netscape Application Server

How Users Access the Data Sheet

In the DataSheet application, make sure that the `stageURL` data field points to the right URL. For example, if you point your content store at `http://kimba.mcom.com/CS`, then the `stageURL` field must point to a folder on that server, such as `http://kimba/docs`. Note that `docs` is the default primary document directory for an Enterprise Server.

In addition, when you build a sample data sheet in Process Express, PAE stores the data sheet in a folder specific for each process instance. For example, suppose you have a default Windows NT installation. If your process is titled "AS2015-test" and its ID is 58, then the new data sheet is stored here:

```
C:\Netscape\SuiteSpot\docs\CS\DataSheet\58\AS2015-test.html
```

The URL displayed to end users in Process Express is then:

```
http://kimba/CS/DataSheet/58/AS2015-test.html
```

Process Express and Netscape Application Server

If you have Netscape Application Server installed on the same system as Process Express, you will not be able to use Netscape Communicator with file attachments. To avoid this situation, you can install NAS and Process Express on different systems, or you can use Microsoft Internet Explorer to access Process Express.

For additional information about using the File Attachment data field, see "File Attachment" on page 140

Adding an Email Attribute for a User

Some sample applications require a user to have an email account. For example, both the TimeOffRequest and DataSheet applications require the admin user to have email. If this user doesn't have email, you will get an error in the logs.

To add email for a user, follow these steps:

1. Launch Netscape Console.

On Windows NT, you can do this by using the Start/Programs menu command and looking for the Netscape Server Family menu item.

2. In the authentication dialog box, enter this information:
 - administrative user name
 - administrative password

- administration URL for the new directory server's Administration Server, including the port number
3. Click the User and Groups tab.
 4. Enter "admin" or "adm*" in the user field, then click Search.
 5. Select the admin user ID from the list of users displayed. By default, this is the NAS Administrator user.
 6. Click Edit.
 7. Enter an email account. If you are the administrator yourself, and you are just testing the system, enter your own email address to see what email notifications you can receive.
 8. Click OK.
 9. Close the Netscape Console.

Using File Attachments and Content Stores

If you want to deploy and use the DataSheet application (or other applications that use file attachments), you must set a new access control list (ACL) for the content store. You must then define the content store properties in Process Builder. In summary, you must perform the following main steps:

- Step 1: Administer the Enterprise Server
- Step 2: Edit the Access Control List
- Step 3: Assign the Content Store's Style
- Step 4: Set the Values for the Application's Content Store

These steps are described in detail in the sections that follow.

Step 1: Administer the Enterprise Server

You set up an access control list (ACL) through the Enterprise Server. To administer the Enterprise Server, perform the following steps:

1. Go to the Server Administration page for your Enterprise Server.
2. Click the button for your Enterprise Server instance. You'll go to its Server Manager pages.
3. Click the Configuration Styles button.
4. Click the New Style link.
5. Enter `ContentStore` as the Style Name field and click OK.
6. In the page that appears, click the "Edit this style" button.
7. Click the Restrict Access link. This displays the Access Control List Management page.
8. Make sure that "the style `ContentStore`" is displayed, then click the "Edit Access Control" button. This displays the Access Control user interface.

Step 2: Edit the Access Control List

Within the Access Control user interface, perform the following steps:

1. Click the New Line button twice. This creates two new ACL lines.
 - Leave the first line unchanged.
 - In the second line, click Deny. A second pane appears in the lower part of the page.
2. Click Allow in the lower pane.
3. Click Update to update the top pane's values.
4. In the second line, click "anyone".
5. In the lower pane, click the "Authenticated people only" radio button.
6. Click the "All in the authentication database" radio button.

7. Make sure that the Default authentication method is selected.
8. Under “Authentication Database” at the bottom of the pane, select your corporate directory from the dropdown list, then click the radio button next to it.
9. Click Update in this frame to update the top pane.
10. In the top frame, click Submit to set the change.
11. Click OK to save your changes.
12. Click Save and Apply to apply your changes to the server.

Step 3: Assign the Content Store’s Style

After editing the ACL, assign the content store’s style by performing the following steps:

1. Under Configuration Styles, click the Assign Style link.
2. Enter `CS/*` as the URL prefix wildcard. This causes the URL to be:
`http://<yourServer>/CS/*`
3. Select ContentStore from the drop down list and click OK.
4. Click OK again to save your changes.
5. Click Save and Apply to apply your changes to the server.

Step 4: Set the Values for the Application’s Content Store

After working with Enterprise Server (as described in Steps 1, 2, and 3), you must make changes in Process Builder. You must set the content store values by performing the following steps:

1. If necessary, start Process Builder.
2. Locate the ContentStore entry in the application tree view.
3. Open the Inspector window for the ContentStore entry.

4. Enter a URL with the format `http://<yourEnterpriseServer>/CS`.
5. Enter the administrative user and password for the Enterprise Server. For example, enter `admin` and `admin`.

For more information on content store properties, see “The Content Store Inspector Window” on page 156.

The Office Setup Application

This chapter describes the Office Setup sample application that is available with PAE.

This chapter contains the following sections:

- Office Setup Application Overview
- Office Setup Process Map
- Office Setup Walkthrough
- The Office Setup Groups
- Data Dictionary
- Form Dictionary
- Script Dictionary
- Customizing the Appearance of the Forms
- Configuring the Office Setup Application

Office Setup Application Overview

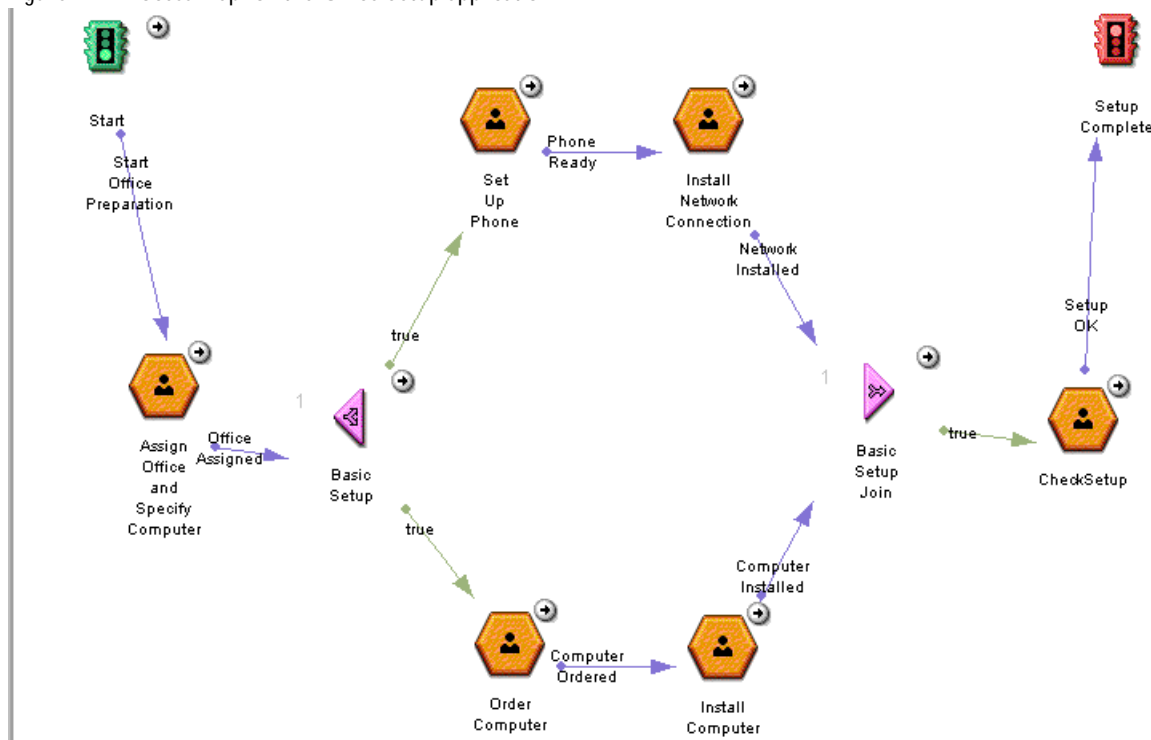
This application controls the process of setting up an office for a new employee. This application illustrates parallel processing.

This application contains a number of tasks that different people must perform to get an office ready. Some of the tasks must be carried out sequentially; for example, the purchasing department must order the computer before the MIS department can install it. Some of the tasks can be carried out in parallel; for example, the MIS department can set up the phone while the purchasing department is ordering the computer.

Office Setup Process Map

The process map for the Office Setup application is shown in Figure 12.1.

Figure 12.1 Process Map for the Office Setup application



Office Setup Walkthrough

This section walks through the complete process, which contains the following entry points, activities (work items), splits, joins, and exit points:

- Start Entry Point
- Assign Office and Specify Computer Work Item
- Basic Setup Split
- Set Up Phone Work Item
- Install Network Connection Work Item
- Order Computer Work Item
- Install Computer Work Item
- Basic Setup Join
- CheckSetup Work Item
- Setup Complete Exit Point

Start Entry Point

The process begins at the Start entry point, as shown in Figure 12.2:

Figure 12.2 The Start entry point



To start the process, the hiring manager fills in an entry point form specifying the new employee's name and start date in the form for the Start entry point, `prepareForNH.html`. This form is shown in Figure 12.3:

Figure 12.3 The prepareForNH.html form

Process Management 

Setup Office for New Employee

New Hire Set Up Request

To see the process map, click [here](#).

Who uses this form:	The hiring manager for a new employee.
When to use this form:	Use this form to start the process for setting up the new employee's office.

Enter information about the new employee:

Employee Information

Name

Start Date: (MM/DD/YYYY)

Report To:

Department:

When you have entered the data about the new employee, press the button below. The next step in the process is for the administrative assistant to assign the office number for the new employee and to specify the computer to be ordered for the employee.

After completing the above form, select one of the following actions.

This form sets the values of the following data fields:

dfEmpName The name of the new employee.

dfStartDate The new employee's start date. This must be in the format MM/DD/YYYY.

dfMgr The hiring manager.

dfDeptName The department that the new employee will be joining. This datafield is a SELECT data field with a menu offering the choices Engineering, Human Resource, Marketing, and Sales.

This entry point has a completion script, `setRequesterField`, that sets the value of the `dfRequesterName` data field based on the process instance creator.

Assign Office and Specify Computer Work Item

The next step is for the manager's administrative assistant to assign an office and specify a computer, as shown in Figure 12.4:

Figure 12.4 The Assign Office and Specify Computer work item



The administrative assistant enters the office location and computer for the new employee, as shown in Figure 12.5. This form sets the values of the following data fields:

dfCubeNo The new employee's office number.

dfFloor The floor containing the office.


dfBldg The building where the office is located.

dfComputer A description of the computer to order for the new employee.

Figure 12.5 The assignCube.html form

Process Management

Setup Office for New Employee



Assign Office Number and Specify Computer for New Employee

Who uses this form:	The administrative assistant for a hiring manager who has a new employee starting soon.
What to do with this form:	Enter information about the cube number and location for the new employee. Also enter a description of the computer to order for the employee.

Here is the information about the new employee:

Employee Information

Name: John Smith

Start Date: 01/19/1999

Report To: Jane Doe

Department: Engineering

Requested By: SuiteSpot Administrator

Please enter the following information about the new employee's office location:

Office Location

Office Number:

Floor:

Building:

Enter information about the computer to order for the employee.

Computer

Computer Description:

Basic Setup Split

At the Basic Setup split, the process splits into two parallel branches, as shown in Figure 12.6:

Figure 12.6 The Basic Setup split



One branch controls the process of installing the phone and network connection. The other branch controls the process of ordering and installing the computer.

The split has two conditional transitions leading off from it. As with decision points and automated activities, you can specify conditions to determine whether a route leading from a split is followed or not. By default, the condition for all routes leading out of a split are set to true, so that they are all active.

When you place a split icon on the process map, the icon splits and appears on the map in two halves. The split part appears where you placed it, and the corresponding join icon appears to the right. The distance between the split and join icons allows for the insertion of two activities between them. When you place the split icon, you may need to scroll your process map to the right to bring the join icon into view.

Each route leading away from a split must ultimately end at the corresponding join. It cannot go to an exit point or to a join for another split. Each branch of the split represents a series of activities that are performed in parallel and ultimately merge back into a single path.

A split icon does not represent an activity, so there are no forms for it.

Set Up Phone Work Item

In the network installation branch, the first step is for the MIS department to assign a phone number and set up the phone. The process map for this step is shown in Figure 12.7:

Figure 12.7 The Set Up Phone work item




The MIS person uses the `setupPhone.html` form, as shown in Figure 12.8. This form sets the values of the following data field:

dfPhone The new employee's phone number.

Figure 12.8 The setupPhone.html form

Process Management

Setup Office for New Employee



Set up Phone

Who uses this form:	MIS personnel.
What to do with this form:	You need to set up the phone for a new employee. When you have finished setting up the form, submit this form.

Here are the details of the new employee:

Employee Name

Name: John Smith

Start Date: 01/19/1999

Report To: Jane Doe

Department: Engineering

Requested By: SuiteSpot Administrator

Here is the information about the employee's office location:

Office Location

Office Number: 1234

Floor: 1

Building: Building 1

Enter the employee's phone number:

Phone Information

Phone:

Install Network Connection Work Item

The next step, also done by the MIS department, is to install the network connection, making sure that the office has network cables and hardware. The process map for this step is shown in Figure 12.9:


Figure 12.9 The Install Network Connection work item



The MIS department uses the `setupNetwork.html` form for this work item, as shown in Figure 12.10. This form sets the values of the following data field:

dfNetworkAddress The network address.

Figure 12.10 The setupNetwork.html form

Process Management 	
Setup Office for New Employee	
Set Up Network Connection	
Who uses this form:	MIS Personnel
What to do with this form:	Submit this form to indicate that you have set up the network connection for a new employee.
Employee Name	
Name: John Smith Start Date: 01/19/1999 Report To: Jane Doe Department: Engineering Requested By: SuiteSpot Administrator	
Office Location	
Office Number: 1234 Floor: 1 Building: Building 1	
Phone Information	
Phone: (555) 555-5555	
Network Information	
IP Address: <input type="text"/>	
<p style="text-align: center;"><i>After you have installed the network connection press the button below to submit this form. The computer installation will be going on in parallel with the network installation. When</i></p>	

Order Computer Work Item

There is no need to wait for the phone and network setup to be finished before ordering and installing a computer in the office. Thus a parallel branch of the process controls the purchase and installation of the computer. In the computer installation branch, the first step is for someone in the purchasing department to order a new computer. The process map for this step is shown in Figure 12.11:

Figure 12.11 The Order Computer work item



This work item has the form `orderComputer.html`, as shown in Figure 12.12. This form sets the values of the following data fields:


dfCompOrderDate The date on which the computer was ordered. This value must be a date such as 12/12/1998. This value is set by a client-side script, but the person ordering the computer can overwrite it. For more information on the client-side script, see “Embedded Client-Side Script” on page 272.

dfCompOrderID The ID or PO number for the order.

Figure 12.12 The orderComputer.html form

Process Management

Setup Office for New Employee



Order Computer

Who uses this form: Purchasing department.

What to do with this form: Use this form to indicate that you have ordered a computer for a new employee.

Here are the details of the new employee:

Employee Name

Name: John Smith
 Start Date: 01/19/1999
 Report To: Jane Doe
 Department: Engineering
 Requested By: SuiteSpot Administrator

Here is information about the employee's office:

Office Location

Office Number: 1234
 Floor: 1
 Building: Building 1

The hiring manager should have entered the computer description. If the description field is blank, please contact the hiring manager. Please order the computer and enter the order date and order ID number.

Computer

Order Date: (MM/DD/YYYY)
 Order ID:
 Description:

After you have ordered the computer press the button below to submit this form. The next step in the

Install Computer Work Item

The next step is for the MIS department to install the computer. The process map for this step is shown in Figure 12.13:

Figure 12.13 The Install Computer work item




This work item uses the `installComp.html` form, as shown in Figure 12.14. This form does not set any data field values. Its purpose is to enable the MIS department to specify when the computer is installed so that the office setup process can progress to the next step.

Figure 12.14 The installComp.html form

Process Management

Setup Office for New Employee



Install Computer

<i>Who uses this form:</i>	MIS Personnel
<i>What to do with this form:</i>	You need to install a computer for a new employee. When the computer is installed, submit this form.

Here are the details of the new employee:

Employee Name

Name: John Smith
 Start Date: 01/19/1999
 Report To: Jane Doe
 Department: Engineering
 Requested By: SuiteSpot Administrator

Here is information about the employee's office:

Office Location

Office Number: 1234
 Floor: 1
 Building: Building 1

Here is the information about the computer to be installed.

Computer

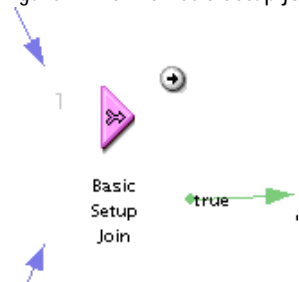
Order Date: 01/19/1999
 Order ID: 12345
 Description: NT 4.0 6 gig hard disk

After you have installed the computer press the button below to submit this form. The network installation will be going on in parallel with the computer installation. When both

Basic Setup Join

When the network setup and computer installation is finished, the two parallel branches merge back together into the main branch of the process at the join. The process map for this step is shown in Figure 12.15:

Figure 12.15 The Basic Setup join

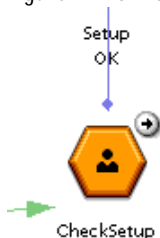


As with a split, the transitions leading away from a join can be conditional. The default condition for each transition is true, and the first transition evaluated that is true is the one followed for a particular process instance.

CheckSetup Work Item

The final step is for the administrative assistant to double-check that the office has been set up properly and has a chair and new package. The process map for this step is shown in Figure 12.16:

Figure 12.16 The CheckSetup work item




This work item has the `checkSetup.html` form, as shown in Figure 12.17.

Figure 12.17 The checkSetup.html form

Process Management

Setup Office for New Employee



Check Office Setup

Who uses this form:	Administrative assistant for a hiring manager who has a new employee starting soon.
What to do with this form:	<p>When you receive this work item in your work item list it means that the computer for the new employee has been installed.</p> <p>You need to check that the installation was correct -- check that the right kind of computer was installed in the right office.</p> <p>If the set up is correct, put a chair and a "New Hire" stationery kit in the employee's office, then submit this form.</p>

Here are the details of the new employee:

Employee Name

Name: John Smith
 Start Date: 01/19/1999
 Report To: Jane Doe
 Department: Engineering
 Requested By: SuiteSpot Administrator

Here is information about the employee's office location:

Office Location

Office Number: 1234
 Floor: 1
 Building: Building 1

Here is the information about the computer:

Computer

Order Date: 01/19/1999
 Order ID: 12345

This work item has a completion script, `verifySetup`, that ensures that the administrative assistant has checked “yes” for the following entries:

Chair in office? This is the `dfChairPresent` radio button data field.

New Hire Stationary Packet on desk? This is the `dfNewHirePacket` radio button data field.

Computer starts correctly? This is the `dfComputerOK` radio button data field.

Setup Complete Exit Point

This application has one exit point, which is reached after the administrative assistant successfully submits the Check Setup form. The exit point is shown in Figure 12.18:

Figure 12.18 The Setup Complete exit point



This exit point has a notification that sends mail to the process instance creator telling them that the process instance has finished. The body of the notification is constructed by the `buildExitNotification` toolkit script.

The Office Setup Groups

This application contains the following application groups:

Purchasing The purchasing department performs the Order Computer work item.

Admin Assistant The administrative assistant performs the Assign Office and Specify Computer work item, as well as the CheckSetup work item.

MIS Dept The MIS department performs the Set Up Phone, Install Network Connection and Install Computer work items.

admin This is the general administrative group.

To make it easier to install and use the sample application in “test” mode, a single `admin` user has been put into each of these groups. If you want to add additional users or change the user, see “Configuring the Office Setup Application” on page 274 for more information.

Data Dictionary

The data dictionary has the following data fields:

dfBldg The building where the new employee's office is located. This is set by the administrative assistant in the Assign Office and Specify Computer work item.

dfChairPresent This is a radio button with choices “yes” and “no” to indicate whether or not the administrative assistant has checked that a chair is present in the new employee's office. This value is set by the administrative assistant in the Check Setup work item.

dfCompOrderDate The date on which the new computer was ordered. This value is set automatically by a client-side script in the Order Computer work item.

dfCompOrderID The order number for the new computer. This value is set by the purchasing department in the Order Computer work item.

dfComputer A description of the computer to order for the new employee. This is set by the administrative assistant in the Assign Office and Specify Computer work item.

dfComputerOK This is a radio button with choices “yes” and “no” to indicate whether or not the administrative assistant has checked that the computer in the new employee's office starts up. This value is set by the administrative assistant in the Check Setup work item. (Although the MIS personnel who installs the computer would be expected to check the installation, the administrative assistant also starts it once to do a double check.)

dfCubeNo The new employee's office number. This is set by the administrative assistant in the Assign Office and Specify Computer work item.

dfDeptName The department that the new employee will be joining. This is set by the hiring manager in the entry point.

dfEmpName The new employee's name. This is set by the hiring manager in the entry point.

dfFloor The floor where the new employee's office is located. This is set by the administrative assistant in the Assign Office and Specify Computer work item.

dfMgr The new employee's manager. This is set by the hiring manager in the entry point.

dfNetworkAddress The IP address for the new employee's network connection. This is set by the MIS department in the Install Network Connection work item.

dfNewHirePacket This is a radio button with choices "yes" and "no" to indicate whether or not the administrative assistant has put a new hire stationery packet in the new employee's office. This value is set by the administrative assistant in the Check Setup work item.

dfPhone The phone number for the new employee. This is set by the MIS department in the Set Up Phone work item.

dfRequesterName The name of the person requesting the office setup. This would usually be the same as the new employee's hiring manager. This is set by the hiring manager in the entry point. This data field would be useful to keep track of the requestor in cases where someone other than the hiring manager is starting the process.

dfStartDate The start date for the new employee. This is set by the hiring manager in the entry point.

Form Dictionary

The application uses the following forms:

- `assignCube.html`
- `checkSetup.html`
- `orderComputer.html`
- `installComp.html`
- `prepareForNH.html`
- `setupNetwork.html`
- `setupPhone.html`
- `status.html`

All of these forms except the `status.html` form are described in “Office Setup Walkthrough” on page 249.

The `status.html` form, shown in Figure 12.19, is used to show the status of the office setup to interested parties. It shows the data gathered on the other forms in the areas of new employee details, office location, phone information, network information, and computer. It is available to members of “all” beginning with the Assign Office and Specify Computer step and throughout the rest of the process.

Figure 12.19 The status.html form

Process Management			
Setup Office for New Employee			
Set Up Office for New Hire Process			
<i>Who uses this form:</i>		Anybody who wants to look at the progress of the cube set up for a new employee.	
New Employee Details			
Name: John Smith			
Start Date: 01/19/1999			
Reports To: Jane Doe			
Department: Engineering			
Office Set up Requested By: SuiteSpot Administrator			
Office Location			
Office Number:	1234		
Floor:	1		
Building:	Building 1		
Phone Information			
Phone:	(555) 555-5555		
Network Information			
IP Address:	888.88.888		

Script Dictionary

The process uses customized scripts of the following kinds:

- Completion Scripts
- Toolkit Scripts

The process also uses one client-side script to automatically set the order date when a new computer is ordered.

- Embedded Client-Side Script

Completion Scripts

This process uses two completion scripts: `setRequesterField` and `verifySetup`.

- `setRequesterField`
- `verifySetup`

`setRequesterField`

This runs as a completion script on the entry point. It checks the Directory Server for the person that initiated the request, and puts their common name as the value of the `dfRequester` field. If the creator is not found in the Directory Server, the `dfRequester` field is not set. The purpose of setting this field is to store the process instance creator in a data field.

The code for this script is as follows:

```
function setRequesterField(){
    // Get the process instance.
    var pi = getProcessInstance();

    // Get the distinguished name of the creator of the
    // process instance from the Directory Server.
    var creator = pi.getCreatorUser();
    // If we can't find the creator in the Directory Server, log
    // an error, but it's not a show-stopping error so keep going.
    if (creator == null || creator == "undefined") {
```

```

        logErrorMsg("creator can not be located");
    }
    // If we found the creator in the Directory Server
    // put their common name in the dfRequester data field.
    else {
        var creatorName = creator.cn;
        pi.setData("dfRequesterName", creatorName);
    }
    // Return true so that the activity succeeds
    // and the process continues.
return true;
}

```

verifySetup

This runs as a completion script on the Check Setup work item. This script checks that the dfChairHere, dfComputerOK, and dfNewHirePacket radio button data fields are all set to “yes.” If any of them are not set to “yes,” the completion script returns false, which means the work item cannot be completed.

The plan is that the administrative assistant checks these buttons after checking that the new employee's office has a chair, the computer starts up, and a new hire stationery packet is on the desk.

The code for this script is as follows:

```

function verifySetup(){
var pi = getProcessInstance();
// Return false if any of the dfChairPresent, dfNewHirePacket
// or dfComputerOK data fields are not set to yes.
if ((pi.getData("dfChairPresent") != "yes") ||
    (pi.getData("dfNewHirePacket") != "yes") ||
    (pi.getData("dfComputerOK") != "yes"))
{
// Tell the user why the form could not be submitted.
setErrorMessage("You must ensure that the computer is OK, " +
+ "and there is a chair and new hire packet in the office.");
return false;
}
// If everything's OK, return true.
return true;
}

```

Toolkit Scripts

There is one toolkit script, `buildExitNotification`.

`buildExitNotification`

This script builds a string to use as the body of an email that gets sent by the exit point to the process instance creator.

This email informs the process instance creator that the process instance has terminated.

The code is as follows:

```
function buildExitNotification(){
    // Get the process instance.
    var pi = getProcessInstance();

    // Get the current work item.
    var wi = getWorkItem();
    var nodeCN = wi.getCurrentActivityCN();

    // Construct a string to use as an email body such as:
    // "The office setup request you initiated on 10/10/ 1998 for
    // new hire Nikki Beckwell has finished at the Setup Complete
    // exit point.

    var body = "The office setup request you initiated on "
        + pi.getCreationDate()
        + " for new hire "
        + pi.getData("dfEmpName")
        + " has finished at the "
        + nodeCN
        + " exit point.";
    return body;
}
```

Embedded Client-Side Script

The `orderComputer.html` form uses a client-side script to automatically set the date on which the computer is ordered. The purchasing department can override this date.

This script makes use of the fact that every data field shown in the form has a corresponding form element of the same name. Thus by setting the value of the `dfCompOrderDate` form element, we are effectively setting the value of the `dfCompOrderDate` data field.

```
function getDate( )
{
    var currDate = new Date();
    /* Get the current year.

NOTE: according to the JavaScript Reference,
The getYear method returns either a 2-digit or 4-digit year:
For years between and including 1900 and 1999, the value returned
by getYear is the year minus 1900. For example, if the year is 1976,
the value returned is 76.

For years less than 1900 or greater than 1999, the value returned
by getYear is the four-digit year. For example, if the year is 1856,
the value returned is 1856. If the year is 2026, the value returned
is 2026.
*/

    var currYear = currDate().getYear();
    if( currYear < 100 )
        currYear += 1900;

    // Jan is 0, Feb is 1 etc... so add 1 to the month.
    var dateString = (currDate.getMonth() + 1) + "/"
        + currDate.getDate() + "/"
        + currYear;

    return dateString;
}

// Put the current date in the dfCompOrderDate
// element in the form.
document.forms[0].dfCompOrderDate.value = getDate();
```


Customizing the Appearance of the Forms

The forms used in this sample application were edited in an external HTML editing tool to fine-tune their appearance.

All HTML forms link to a single style sheet, `mystyles.html`, that defines a class that is used to define the purple border for the paragraph at the bottom of each form.

The html pages for forms appear at the top level of the subdirectory of Applications that contains your application.

To edit these forms, use an HTML editor that meets the following requirements:

- The editor must ignore all tags and attributes it does not understand, because the data fields are implemented as customized SERVER tags. Some HTML authoring tools will delete tags and attributes that are not standard HTML.
- The editor must leave the following tags upper case: HEAD, BODY, HTML, and SERVER.

HTML editors that meet these requirements include Netscape Communicator (versions 4.5 or later) and Microsoft Internet Explorer (versions 4.0 and later).

If you edit the forms in an external editor, be sure not to modify any of the customized SERVER tags, or you the data fields on the forms will not work.

If you edit a web page for a form while it is also open in Process Builder, be sure to click on the form again in Process Builder before saving the application. When you click on a form that has been edited externally, Process Builder asks if you want to load the modified form with the latest changes.

Note that the web page for the form only contain a portion of the information the end-user sees on the form. The Process Engine inserts the banner, buttons, and comment areas of the form, as well as the `<FORM>` tag itself.

For more information on using an external editor with Process Builder, see “Modifying Forms” on page 165.

Configuring the Office Setup Application

Before you can deploy the Office Setup application, you must make sure your environment meets the following requirements:

- Set the corporate directory as described in “Setting Your Corporate Directory” on page 70.
- Make sure that at least one valid user is present in each of the following groups: Purchasing, MIS Dept, admin, and Admin Assistant.

Typically, you don't need to add users to the above groups, because an `admin` user is added to each of those groups by default. However, you must add users if:

- You do not have an `admin` user in your corporate database.
- You would like to use someone other than the `admin` user or in addition to the `admin` user.

To add users, perform the following steps:

1. Make sure that you have defined a corporate directory for the application. For more information, see “Applications and the Corporate Directory” on page 69.
2. In Process Builder, open the application tree view.
3. In the Groups and Roles folder, right-click the group name you want to change, and then choose Properties.
4. Using the Browse option, find the users you want to add, and drag them to the List of users. Or using the Search option, highlight the users and click Add.
5. Close the dialog box.

After you have saved the application, you are ready to deploy it.

The Loan Management and Credit History Applications

This chapter describes the Loan Management sample application and its subprocess, the Credit History application. These two sample applications show how to use a subprocess in your applications.

This chapter includes the following sections:

- Loan Management Application Overview
- Credit History Application Overview
- Loan Management Process Map
- Credit History Process Map
- Loan Management and Credit History Walkthrough
- Groups and Roles
- Data Dictionary
- Loan Management Script Dictionary
- Credit History Script Dictionary
- Configuring the Loan Management Application
- Configuring the Credit History Application

Loan Management Application Overview

The Loan Management application controls the process of approving a loan, from the first request to the final approval.

As part of the process, this application contains a subprocess step (Check Credit History) which launches the subprocess application Credit History. This application checks to see how much credit the person applying for the loan has been extended in the past, and returns the information to the Loan Management application. The Loan Management and Credit History applications are specifically designed to show how subprocesses work.

Note that the names for the Check Credit History subprocess and Credit History application are slightly different to distinguish the two. Check Credit History refers to the subprocess icon on the Loan Management process map and to the subprocess item's properties. Credit History refers to the actual application that is called by Loan Management as a subprocess.

Credit History Application Overview

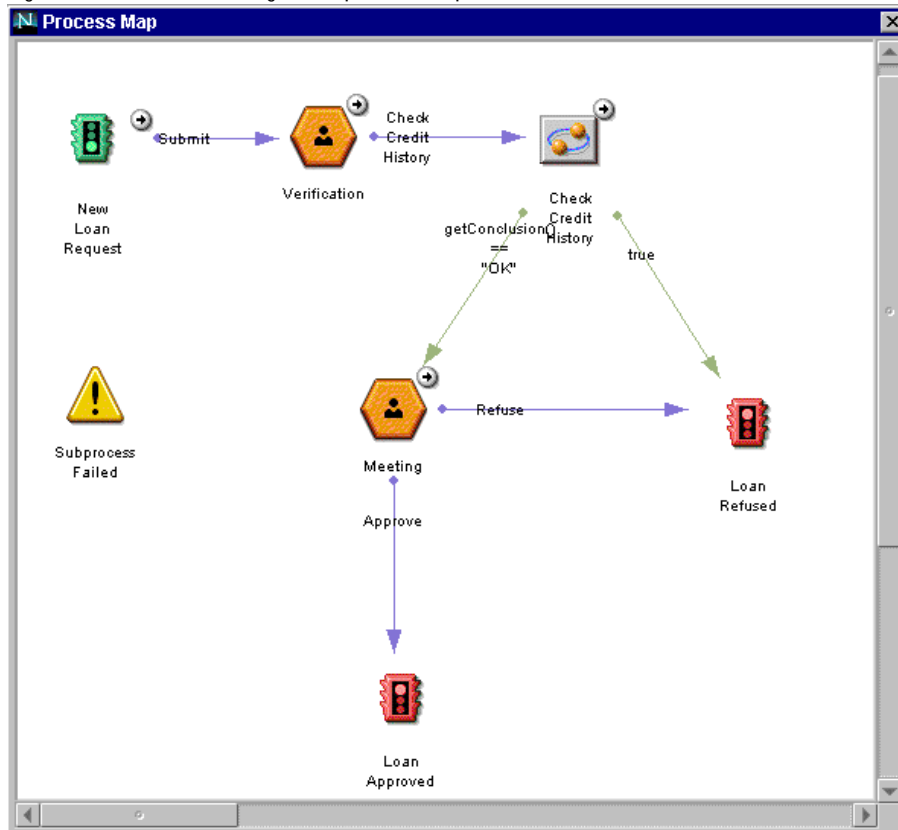
The Credit History application can be run either as a subprocess of the Loan Management application or as a stand-alone application. It shows the process of checking a customer's previous credit. In the real world, that check would probably be done against a database of customer history, but for the purpose of this sample application the amount is hardcoded into the application.

Though the entry point for the Credit History application appears on the process map, it is not included in the walkthrough portion of this chapter. The entry point is only seen by a user if the user runs the Credit History application as a stand-alone application, instead of as a subprocess. Because the walkthrough shows the Credit History application called as a subprocess, the entry point is handled automatically when the Loan Management application starts the subprocess.

Loan Management Process Map

Figure 13.1 shows the process map for the Loan Management application:

Figure 13.1 The Loan Management process map

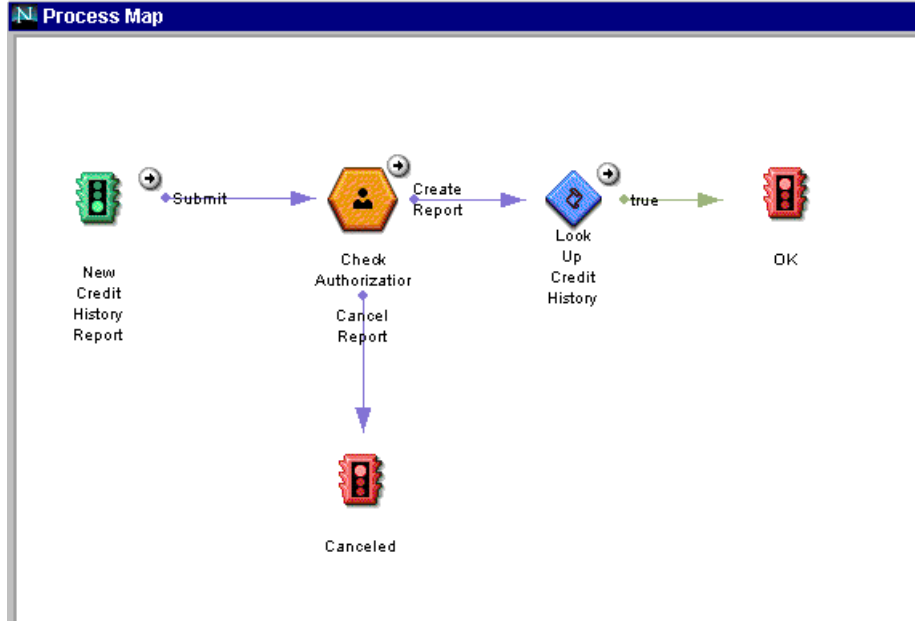


Credit History Process Map

Figure 13.2 shows the process map for the Credit History application.

If run as a stand-alone application, the user starts at the New Credit History Report entry point. If run as a subprocess, the New Credit History Report Entry point happens automatically when this application is called from the Loan Management application.

Figure 13.2 The Credit History process map



Loan Management and Credit History Walkthrough

This section walks through the complete process, including both the Loan Management and Credit History applications, step by step. The complete process contains the following entry points, work items (activities), subprocesses, and exit points:

- New Loan Request Entry Point
- Verification Work Item
- Check Credit History Subprocess
- Credit History Application as a Subprocess
- Check Authorization Work Item
- LookUp Credit History Automated Activity

- Canceled Exit Point
- OK Exit Point
- Meeting Work Item (Parent Process)
- Loan Refused Exit Point
- Loan Approved Exit Point

New Loan Request Entry Point

The Loan Management process starts when a user puts in a new loan request. This user is someone at a bank who is responsible for helping customers with their loan applications.

Figure 13.3 The New Loan Request entry point



To start a new loan request, the person at the bank accesses the `creation.html` form in Process Express, as shown in Figure 13.4.

Figure 13.4 The creation.html form

The screenshot shows a web form titled "creation.html". At the top, there is a box labeled "Application Banner" with instructions: "Customize this banner by importing an image called **banner.gif** in the <application root>/**images** folder using file | import in the Builder Tool". Below this, there are three input fields: "Title:" followed by a text box, "Customer:" followed by a text box and a grey "Address Book" button, and "Amount:" followed by a text box. A large, faint "Test mode" watermark is visible across the form. Below the form, there is a blue bar containing a grey "Submit" button. Below the bar, the text reads: "After completing the above form, select one of the following actions."

This form sets the values of the following data fields:

title The title of the process instance. Since it appears on the work list, this field needs a value that is meaningful to the end users, for example, the kind of loan requested.

customer The name of the customer who is requesting the loan. This is a user picker widget. The customer name is used for the field role Customer. This user needs to be in the corporate directory.

amount The amount of money the loan is for.

Verification Work Item

The next step is for the creator of the loan application to verify the information entered to make sure that it is correct. The process map for this step is shown in Figure 13.5.

Figure 13.5 The Verification work item



This step uses the same form as the entry point, `creation.html`, shown in Figure 13.4. The only additional field is the “comments” text area, where the creator can enter comments that will appear in the details and history. The comments section appears because the Allow to Add Comments property for the Verification activity is set to true.

Check Credit History Subprocess

At this step the Loan Management application calls a subprocess, the Credit History application. The process map for this step is shown in Figure 13.6:

Figure 13.6 The Check Credit History subprocess



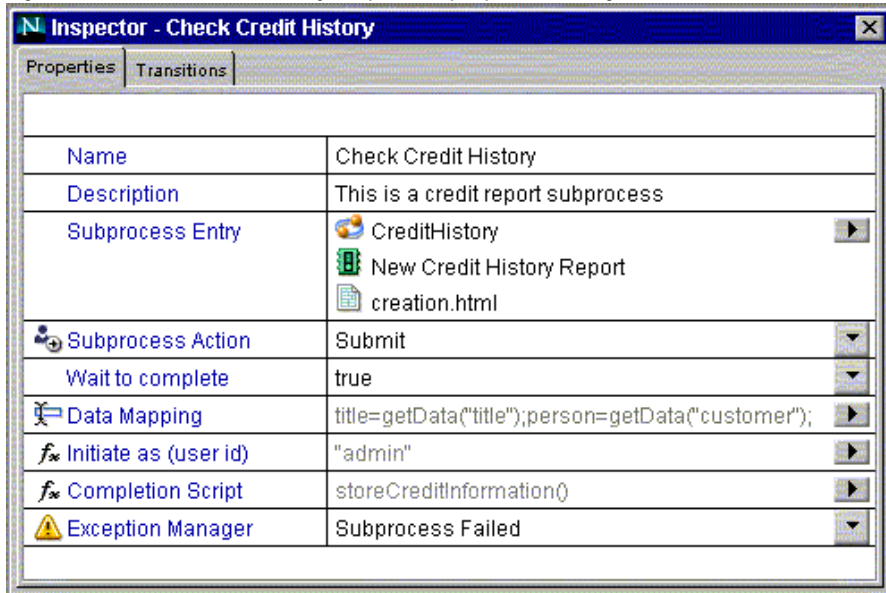
This subprocess checks the credit history of the customer. If the subprocess terminates in the OK exit point, the loan request moves forward to the next step. If the Credit History application does not end at the OK exit point, the loan is denied.

Check Credit History Subprocess Properties

The Check Credit History subprocess in Loan Management has the properties as shown in Figure 13.7. (You can view properties by right-clicking the subprocess item in the process map and choosing Properties.)

The end user does not see these properties, but they govern how the Loan Management application starts the Credit History application as a subprocess.

Figure 13.7 Check Credit History subprocess properties dialog box



The Subprocess Entry shows that the subprocess application is CreditHistory, that the entry point accessed for this application is the New Credit History Report entry point, and that the form at this entry point is creation.html. A subprocess must use an entry point that only has one form. The fields on creation.html that would ordinarily be filled in by the user at the entry point are filled in automatically using data mapping. For more information, see “Data Mapping” on page 283.

The Subprocess Action is Submit, which is the transition that leads from the entry point to the first activity on the Credit History process map.

The Initiator User ID is the ID of the user who initiates a subprocess. In many cases this field would use a script to determine the user by process instance. However, in this sample it is hardcoded to the static value of the admin user. If you are not using the admin user, or want a different user, you can replace this value. For information about revising the defined users, see “Configuring the Credit History Application” on page 297.

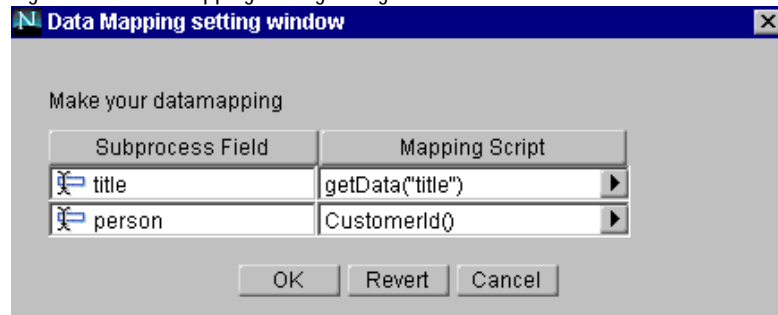
The Data Mapping, Completion Script, and Exception Manager properties are covered in the following sections.

The Transitions tab shows the order in which to evaluate the transitions out of the subprocess. First, the application evaluates the transition named `getConclusion() == "OK"`. This transition checks if the subprocess ended at the OK exit point. If that is true, the process continues to the Meeting work item. If the subprocess did not end at the OK exit point, the next transition is evaluated. Because the next transition is `true`, it acts as an “else” statement. If the subprocess ended anywhere except the OK exit point, the process follows this transition to the Loan Refused exit point.

Data Mapping

In the Data Mapping dialog box, the parent process fields are mapped to the subprocess fields that are required to complete the form at the subprocess’s entry point. Figure 13.8 shows the dialog box for this sample application.

Figure 13.8 Data Mapping setting dialog box



In this sample application, the `creation.html` form (the form at the entry point of the Credit History application) contains two fields that must be filled in: `title`, and `person`.

The `title` field in the subprocess is mapped to the value in the parent process’s `title` field. The script `CustomerId()` populates the subprocess data field `person` with the user ID of the customer stored in the parent process’s `customer` field. For more information on this script, see “CustomerId Toolkit Script” on page 295.

Completion Script

The completion script `storeCreditInformation` takes the values from the Credit History application and stores them in the appropriate data fields in the Loan Management application. It also determines at what exit point the subprocess ended. For more information, see “`storeCreditInformation` Completion Script” on page 293.

The Subprocess Failed Exception Manager

If the subprocess fails for some reason, the application calls an exception manager named Subprocess Failed. The tree view shows this exception manager as a yellow triangle. It also appears on the process map, as shown in Figure 13.9, but it is not connected by transitions to any other items.

Figure 13.9 Exception manager



Subprocess
Failed

The exception manager is similar to an activity, in that you assign it to someone (usually the administrator), and that person can view a form and take action. For the Loan Management application, the exception manager is assigned to the creator by default, and the assignee views the `create.html` form.

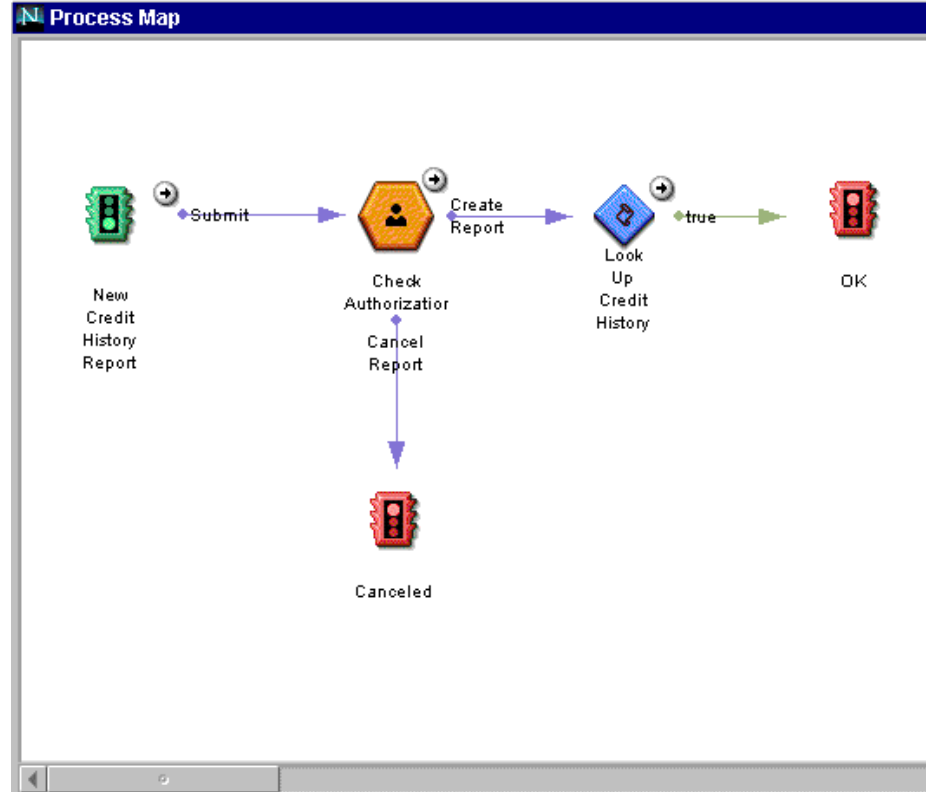
The Subprocess Failed exception appears on a work list only if the subprocess fails. In most cases it won't appear.

Credit History Application as a Subprocess

When the process reaches the Check Credit History step, the Credit History application is started as a subprocess of the Loan Management application.

The process map for the Credit History application is shown in Figure 13.10. The process map displays the steps that take place before returning to the Loan Management parent process.

Figure 13.10 The Credit History process map

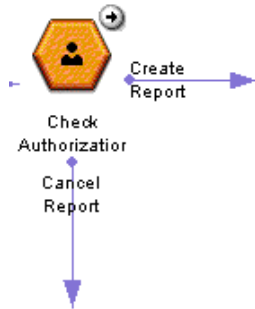


When Loan Management calls Credit History, the New Credit History Report entry point happens automatically. The fields in the form `creation.html` are filled in by data mapping, and the Submit action is performed automatically.

Check Authorization Work Item

The next step in the process is the first activity in the Credit History application, Check Authorization. The process map for this step is shown in Figure 13.11.

Figure 13.11 The Check Authorization work item



This work item is completed by the person responsible for authorizing credit reports. However, for the purposes of the sample application, this work item is assigned by default to the creator. The form associated with this step is `form.html`. This form contains a read-only field, `person`, which is the field role associated with the customer requesting the loan. This field role is picked up from the parent process, Loan Management. In Figure 13.12, the request has been made for the user's SuiteSpot Administrator.

Figure 13.12 The `form.html` form

Application Banner
 Customize this banner by importing an image called **banner.gif** in the `<application root>/images` folder using `file | import` in the Builder Tool

A credit history has been requested for **John Smith**. Do you authorize it ?

You can add additional comments here that will appear in the Details & History page:

After completing the above form, select one of the following actions.

Cancel Report

Create Report

The choices available from this step are Cancel Report and Create Report.

If you choose Cancel Report, the Credit History application continues to the Canceled exit point. If you choose Create Report, the process continues to the automated activity LookUp Credit History.

LookUp Credit History Automated Activity

The next step is the automated activity Look Up Credit History. The process map for this step is shown in Figure 13.13.

Figure 13.13 The Look Up Credit History automated activity



In a real-world situation, this automated activity would look up a customer's credit history in a database. However, for the purpose of this sample application, the values are hardcoded. The value of the data field `credit_history` is set to 10000.95. The Loan Management process uses this amount, along with the date, when the Credit History subprocess returns to the Loan Management application.

After this automated activity runs, the process continues to the OK exit point for the Credit History application.

Canceled Exit Point

If the report is canceled at the Check Authorization step, the process goes to the Canceled exit point, shown in Figure 13.14.

Figure 13.14 The Canceled exit point



Canceled

This step displays the form `refused.html`. This form contains no fields, just the message that the credit history report is not authorized. From this exit point, the subprocess returns to the Loan Management process and continues to the Loan Refused exit point.

OK Exit Point

If the request for the report is approved, the process goes to the OK exit point, shown in Figure 13.15.

Figure 13.15 The OK exit point



OK

This step displays the form `completed.html`, shown in Figure 13.16.

Figure 13.16 The completed.html form

Application Banner
Customize this banner by importing an image called **banner.gif** in the
<application root>/images folder using file | import in the Builder Tool

Credit History Report

User: John Smith

Credit: 10000.95

Credit Last Modified: 01/15/1999 12:41:20

This form shows the information gathered by the `LookUpCreditHistory` script. It has the following fields:

person The person for whom the credit history is gathered. The same as customer in the Loan Management application.

credit_history The amount of money the person has been loaned in the past.

credit_date The date the credit history check was performed.

From this exit point, the subprocess returns to the Loan Management process and continues to the Meeting work item.

Meeting Work Item (Parent Process)

The Check Credit History subprocess has a completion script, `storeCreditInformation`. If this script finds that the subprocess concluded at the OK exit point, the subprocess proceeds to the Meeting step, shown in Figure 13.17.

Figure 13.17 The Meeting work item



In the Meeting step, the person making the loan meets with the customer to review the loan contract's details. For this sample application, the Meeting activity is assigned by default to the creator of the process instance.

At this point, the customer can either approve or refuse the terms. If the customer approves the terms, the process continues to the Loan Approved exit point. If the customer refuses the terms, the process continues to the Loan Refused exit point.

The form displayed at this step is `summary.html`, shown in Figure 13.18.

Figure 13.18 The summary.html form

The screenshot shows a web form with the following elements:

- Application Banner:** A box with a red header "Application Banner" and text: "Customize this banner by importing an image called **banner.gif** in the <application root>/images folder using file | import in the Builder Tool".
- Data Fields:** Read-only text fields for:
 - title: car loan
 - customer: John Smith
 - amount: 4
 - credit: 10000.95 (since 01/15/1999 12:41:20)
- Comments:** A text area with the prompt "You can add additional comments here that will appear in the Details & History page:".
- Actions:** A purple bar containing two buttons: "Accept" and "Refuse".

Based on the results of the meeting, the two actions available to the person making the loan are Accept or Refuse. Clicking Accept sends the process to the Loan Approved exit point, whereas clicking Refuse sends the process to the Loan Refused exit point.

Note that this form is also used throughout the process for the group "all," so that everyone can see a summary of the loan process. This form is also used at the "Subprocess Failed" exception manager.

This form contains the following read-only data fields. They are the data fields from the original loan creation form, along with information gathered in the subprocess:

title The title of the process instance. Since it appears on the work list, this field needs a value that is meaningful to the end users, for example, the kind of loan requested.

customer The name of the customer who is requesting the loan. This is a user picker widget. The customer name is used for the field role Customer. This user needs to be in the corporate directory.

amount The amount of money the loan is for.

credit The amount of credit the customer has had, as determined by the Check Credit History subprocess.

last_modified The date the credit was checked.

Loan Refused Exit Point

If the customer refused the loan terms at the Meeting step, or if the Credit History process ended at the Canceled exit point, the next step is the Loan Refused exit point, shown in Figure 13.19.

Figure 13.19 The Loan Refused exit point



Loan
Refused

The form for the exit point is `summary.html`, shown in Figure 13.18. However, since the process is at an exit point, the fields are display-only, and no action can be taken.

Loan Approved Exit Point

You reach the Loan Approved exit point, shown in Figure 13.20, if the loan is approved in the Meeting step.

Figure 13.20 The Loan Approved exit point



Loan
Approved

The form for the exit point is `summary.html`, shown in Figure 13.18. However, since the process is at an exit point, all the fields are display-only, and no action can be taken.

Groups and Roles

The Loan Management application has the following non-default role:

customer A field-based role representing the customer who requested the loan. The data field `customer` is mapped to the `person` data field in the Credit History application, and so has the same value.

The Credit History application has the following non-default groups and roles:

person A field-based role representing the person who requested the loan. The data field `person` is mapped to the `customer` data field in the Loan Management application, and so has the same value.

trusted users An application group that is composed of users trusted by the application to start the application as a subprocess. These user IDs are the App User IDs of the parent processes that would call the process as a subprocess. For the Credit History application, this group by default contains one user, “admin,” which is the default App User ID for the Loan Management application. You may have changed this value when you configured the Loan Management application. For more information on trusted users, see “The Corporate Group Dialog Box” on page 121.

For more information about configuring the groups and roles for these sample applications, see “Configuring the Loan Management Application” on page 296 or “Configuring the Credit History Application” on page 297.

Data Dictionary

The Loan Management application has the following data fields in its data dictionary:

amount The amount of the loan that the customer is applying for.

credit The amount of credit the customer was last given, based on the credit history.

customer The name of the customer who is requesting the loan. This is a user picker widget. The customer name is used for the field role Customer. This user needs to be in the corporate directory.

last_modified The date the credit history was last modified.

title The title of the process instance. Since it appears on the work list, this field needs a value that is meaningful to the end users, for example, the type of loan requested.

The Credit History application has the following fields in its data dictionary:

credit_date The date the credit report was last updated.

credit_history The amount of credit the customer was last given.

person The user ID of the person whose credit history is requested.

title The title of the process instance. Since it appears on the work list, this field needs a value that is meaningful to the end users, for example, the kind of credit history requested.

Loan Management Script Dictionary

The Loan Management application contains two scripts:

- storeCreditInformation Completion Script
- CustomerId Toolkit Script

storeCreditInformation Completion Script

This script runs after the subprocess is complete. It finds out which exit point the subprocess reached. The script also gets the values of the fields set in the subprocess (`credit_history` and `credit_date`) and sets the values of the data fields in the parent process (`credit` and `last_modified`) to those

values. Notice that this script takes advantage of the `getSubProcessInstance` function to get the subprocess instance data and state. The script's code is as follows:

```
function storeCreditInformation(){
    // Get Handle on subprocess instance
    var spi = getSubProcessInstance();

    // Get Handle on process instance
    var pi = getProcessInstance();

    // getConclusion returns the name of the exit point reached
    // by the subprocess
    switch( getConclusion() ){
        case "OK":
            // The subprocess instance has been successfully completed

            // Read data from the subprocess instance.
            // Please note that "credit_history" and "credit_date"
            // are names of fields defined in the child process.
            var credit_history = spi.getData("credit_history");
            var credit_date = spi.getData("credit_date");

            // Store data in process instance.
            // Please note that "credit" and "last_modified"
            // are names of fields defined in this process.
            pi.setData("credit", credit_history);
            pi.setData("last_modified", credit_date);

            // Success
            return true;

        default:
            // No data transfer required
            return true;
    }
}
```

CustomerId Toolkit Script

This script is used during the data mapping from the parent process to the subprocess. It takes the value of the `customer` field in the Loan Management application and finds the associated user ID for that customer. The user ID is then entered into the `person` field in the subprocess application, Credit History.

This script is needed because field roles are entered as user IDs but are stored as distinguished names. Therefore, the value from the `customer` field must be converted from a distinguished name to a user ID before it can be entered into the `person` field.

The script's code is as follows:

```
function CustomerId(){
    // Get a handle on the process instance;
    var pi = getProcessInstance();

    // Get the customer user.
    var cus = pi.getRoleUser( "customer" );

    // Verify that the role has been populated correctly.
    if ( ( cus == "undefined" ) || ( cus == null ) ){
        logErrorMsg("CUSTOMER_NOT_DEFINED");
        return null;
    }

    // return the user ID of the customer
    return cus.uid;
}
```

Credit History Script Dictionary

The Credit History application has one script, an automation script called `LookUpCreditHistory`.

LookUpCreditHistory Automation Script

This script is used by the automated activity LookUp Credit History. This script determines the values for the `credit_history` and `credit_date` fields. In a real-world situation, this data might be contained in a database. However, for the purpose of this sample application, the values are hardcoded into the script.

```
function LookUpCreditHistory(){
    // Get a handle on the Process Instance.
    var pi = getProcessInstance();

    // Get the information about the user whose
    // credit history is requested.
    var user = pi.getRoleUser("person");
    var userId = user.uid;

    // Based on the userId, determine the
    // credit_history and the credit_date.
    // These values are hardcoded, but we could instead
    // access an external database for more information.
    pi.setData( "credit_history", 10000.95 );
    pi.setData( "credit_date" , new Date() );

    return true;
}
```

Configuring the Loan Management Application

Before you can deploy the Loan Management application, you must make sure your environment meets the following requirements:

- Set the corporate directory as described in “Setting Your Corporate Directory” on page 70.
- Add a user to the admin group.

To add a user to the `admin` group, follow these steps:

1. In the application tree view, right-click the `admin` group and choose Properties. Or highlight the group and click the Inspector button from the toolbar.
2. Using the Browse option, find the users you want to add, and drag them to the List of users. Or using the Search option, highlight the users and click Add.
3. Close the dialog box.

After you have set these properties, you can deploy the Loan Management application. However, you cannot run the application until you also deploy the application it uses as a subprocess, Credit History.

Configuring the Credit History Application

Before you can deploy the Credit History application, you must make sure your environment meets the following requirements:

- Set the corporate directory as described in “Setting Your Corporate Directory” on page 70.
- Add users in two groups: `admin` and trusted users. The `admin` can be anyone, but the default is the `admin` user. The trusted user must be the same user as defined in the Loan Management application for the Credit History subprocess.

To check this user value, perform the following steps:

1. Open the LoanMgmt application.
2. Open the Inspector window for the “Check Credit History” activity.
3. Check the value for the "Initiated As (user id)" property. By default, this is set to `admin`.

To add a user to the admin or trusted user group, perform the following steps:

1. In the application tree view, right-click the group and choose Properties. (You can also double-click the group, or highlight the group and click Inspector from the toolbar.)
2. Using the Browse option, find the users you want to add, and drag them to the List of users. Or using the Search option, highlight the users and click Add.
3. Close the dialog box.

After you have saved the application, you are ready to deploy it.

The Insurance Claim Processing Application

This chapter describes the Insurance Claim Processing sample application that is shipped with this product. It includes a general walk through of the sample application and describes its functions. You can use the applications as a learning tool to see how applications are designed.

This chapter contains the following sections:

- Application Overview
- Process Map
- Application Walkthrough
- Groups and Roles
- Data Dictionary
- Forms
- Script Dictionary
- Required Files
- Configuring the Insurance Claim Processing Application
- Custom Activity Code

Application Overview

This application automates the process of making an insurance claim for automobile damage from the time the customer enters the claim information to the time when the claim is either approved or denied. The application demonstrates how you can implement a number of advanced PAE features including the following:

- Accessing a flat file

The application takes the policy number from the customer and reads a flat file (an XML file with customer records) and parses it to get the policy details record for the customer. If the record is not found the application prompts the user to re-enter the policy number.

- Step by Step wizard like interface

The application uses a step-by-step interface that makes the application simpler to use and understand.

- Expiration for customer forms

The Application uses expiration setter and handler scripts in order to discard forms that are not accessed for a certain time.

- Email notification for claim approval/denial

The application sends email to the person who initiated the insurance claim when the claim is approved or denied.

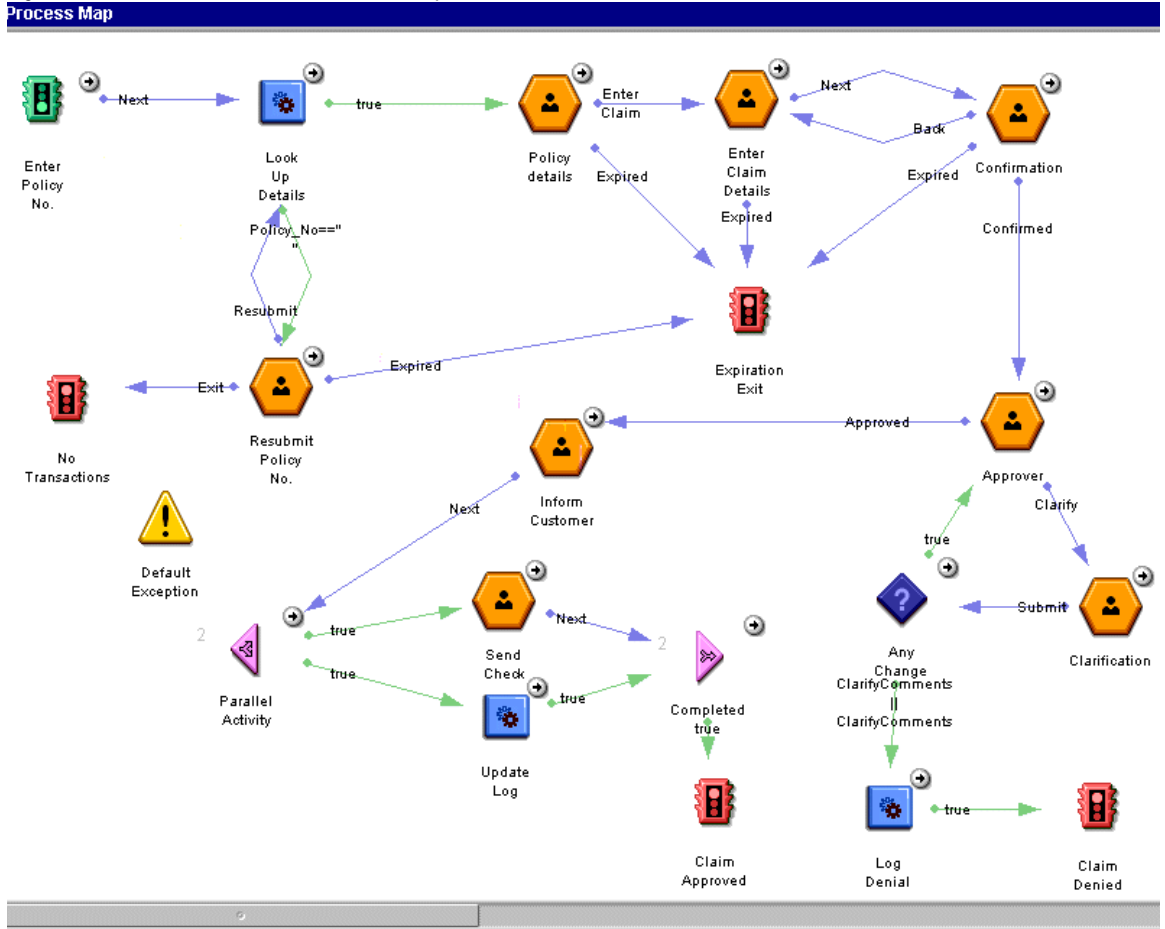
Note The application contains all the necessary forms, user roles, and database fields to complete the process.

You must use Netscape Communicator to run this application.

Process Map

Figure 14.1 shows the application's steps:

Figure 14.1 The Insurance Claim Process Map



The process map shows the following steps.

Entry Point

The process map contains one entry point for entering the Policy number in order to get the record from the flat file database.xml.

Activities

There are eight activities:

- *Policy Details.* Displays the customer personal information and policy details
- *Re-Enter Policy No.* Prompts the user to re-enter the policy number if it was not found in the flat file.
- *Enter Claim Details.* Prompts the user to enter the claim details.
- *Confirmation.* Displays customer personal information, policy details and the claim and prompts for confirmation if everything is correct.
- *Approver.* Approves insurance claim if correct else sends for clarification of the claim
- *Clarification.* Checks for policy changes after talking to the customer.
- *Inform Customer.* Informs the customer that the claim was approved.
- *Send Check.* Displays that the check was sent.

Custom Activities

There are three custom activities, which perform different functions.

Look Up Details

This custom activity is the most important of the three custom activities. It takes the policy number, entered by the customer at the entry point, as the INPUT. It then opens the flat file (database.xml) and parses the file to get the matching policy number as entered by the customer.

If it finds a matching Policy number, it fetches the records and sends all the details to the OUTPUT of the custom activity. All this is done in the perform() function of the custom activity Java file. The record is mapped to the variables in Process Builder and hence when we view the next form we can see all the details.

If it does not find a matching policy number, it prompts the customer to re-enter the policy number until it finds the matching policy number in the flat file.

It maintains a log file (Policy.log) which is created if not present. This log file has the information about all the access to the policy and the results.

Note The flat file (database.xml) must be in the current user directory in order to make the application work.

Update Log

This custom activity updates the Policy.log file whenever the claim is approved. It is associated with a parallel activity.

Log Denial

This custom activity updates the Policy.log whenever the claim is denied.

Decision Point

This decision point checks if the customer details have changed. If there are changes in the policy details, it sends back to the approver else sends for denial. This is because if there are no changes you then the claim is not correct and hence denied.

Parallel Process

The purpose of this parallel activity is to speed the process where we have an activity of Send Check and an Update Log custom activity. This too can be performed parallel as they do not require any manual process.

Exit Points

There are four exit points in the application:

- *Claim Approved.* Exits whenever a claim is approved.
- *Claim Denied.* Exits whenever a claim is denied.
- *Expiration Exit.* Exits whenever an Expiration handler gets activated it destroy the process instance.
- *No Transactions.* Exits when there is no access to the policy details such as when the user exits after entering wrong Policy number.

Notification

There are three notifications in the application. They send email to the customer when:

- A claim is approved.
- A claim is denied.
- A claim process has expired.

Application Walkthrough

This walkthrough takes you through the process described by the application, describing the portions of the application involved in each step.

The Entry Point (Enter Policy No.)

Users (customers) initiate an insurance claim at the Enter Policy No. entry point. When they initiate the application in Process Express, they see the Entry.html form. On this form customer enters the policy number for which they are claiming the insurance.

Custom Activity (Look Up Details)

When users have filled out the form, they click Next to perform the function of the class file associated in the custom activity - Look Up Details. The function of this custom activity is to open `database.xml` (which is the flat file), parse it, and look for a matching policy number.

A main function of this application is to log all the activities carried out whenever a new process is initiated. This helps the insurance company personnel check all insurance claims entered through this application. At this custom activity, the application adds the customer name and a randomly generated claim ID to a file called `Policy.log`.

Activity (Policy Details)

If the custom activity finds a matching Policy number, it displays the policy details on the form `Details.html`. This activity has two scripts associated with it:

- Script at Policy Details Activity
- Completion Script at Policy Details Activity

Activity (Re-enter Policy No.)

If the custom activity does not find the policy number in the flat file, it will prompt the customer to re-enter the policy number until it finds a policy record. It prompts the customer by displaying a form, called `Resubmit.html`.

Activity (Enter Claim Details)

The form at this activity, `EnterClaim.html`, allows the customer to enter the claim. It has predefined claim such as vehicle damage, window damage etc. The customer clicks the appropriate check box to select the type of claim and describes the claim in detail in the text area provided.

This activity also has two scripts associated with it: an Script at Enter Claim Details Activity and a Completion Script at Enter Claim Details Activity.

Activity (Confirmation)

After entering all the details in the EnterClaim.html form, the form at this activity Confirmation.html shows all the details associated with the customer and the policy, including the claim details which the customer had entered in the previous form. This form prompts the customer to check all the details displayed and then continue if correct.

If the customer feels that there is a error in the claim details a 'BACK' button takes the customer back to the Enter Claim Details activity, where the customer can enter the corrects claim details and continue.

This activity also has an Script at Confirmation associated with it.

Activity (Approver)

After the confirmation from the customer, the process continues to the next step, approving the claim. This activity is carried out by the insurance company employee who has the rights to approve the claim. The form Approver.html displays all the information shown previously in the Confirmation.html, the customer personal information, the policy details, and the claim. In addition to this information, the form contains a text area for comments from the approver.

The approver can approve the claim at this point or, if there are some comments or doubts about the claim or the customer, the approver can send it on for clarification.

Activity (Clarification)

If the approver in the previous activity has some doubts about the claim or the customer, the approver can send it for clarification. The form, Clarify.html, has the same contents as Approver.html, and an additional field for the clarification

comments. The clarifier talks with the customer and looks for any changes in the policy. If there are any changes he writes them in the Clarification Comments text area and submits it to the approver.

If there are no changes in the policy, then the approver's comments hold good and there is something wrong with the claim. In this case, the claim is denied.

Activity (Inform Customer)

If the approver approves the claim, this activity displays a confirmation form (InfCustomer.html) that the claim with Customer Name, Policy No. and Address to which the claim check will be sent is approved.

Parallel Activity

After the Inform Customer activity, we have a parallel activity which carries out two different tasks at the same time: Sent Check and Update Log.

Activity (Send Check)

The first part of the parallel activity informs the insurance claim personnel that the check for the approved claim was sent. This is displayed in the SendCheck.html form.

Custom Activity (Update Log)

The other part of the parallel activity is a custom activity - Update log. This custom activity updates the log file, which was appended to when the process was initiated. This custom activity writes the claim ID, which was randomly generated during the process initiation, and the claim, which was approved, in the log file.

Groups and Roles

The following groups and roles are defined for this application:

- all
- creator
- admin
- trusted users

These are the default groups and roles that are created for every application. For demonstration purposes, all activities in this application are assigned to the creator of the process instance. In order for a group to be able to search, the Allow Search option must be checked in the Inspector window for that group.

Group and Role Priorities

The group and role priorities determine the order in which the application processes groups and roles to determine what forms the users see. Because a user can belong to more than one group or role in the same application, the application needs the builder to specify which group or role to evaluate first. The evaluation order you set in the Groups and Roles properties window is the order used in the Form Access window.

The assignee role is always at the top of the list and the “all” group is always at the bottom. You cannot reprioritize these two.

To see the group and role priorities, follow these steps:

1. In the application tree view, double-click the Groups & Roles folder, or highlight the folder and click Inspect.
2. In the Inspector Window, click “evaluation order” if it's not already selected. The groups and roles are listed in the following order:
 - assignee
 - creator
 - admin
 - all

The application first checks to see if a user is the assignee of a particular activity. If so, it finds the appropriate form to display to the assignee. If the user is not the assignee, the application checks down the list, and when it finds a role or group the user belongs to, it displays the appropriate form as configured in the Form Access window. Figure 14.2 shows a portion of this window:

Figure 14.2 Form Access Window

Form Access									
Drag a form from the application tree view to a box in this form.									
	Enter Policy No.	Enter Claim Details	Policy details	Confirmation	Approver	Inform Customer	Send Check	Clarification	Resubmit Policy No.
assignee	X	ClaimEntry.html	Details.html	Confirmation.html	Approver.html	InfCustomer.html	SendCheck.html	Clarify.html	Resubmit.html
admin									
trusted users									
creator	X								
all	Entry.html								

Data Dictionary

The data dictionary contains all the data fields that are used by the application and stored in the database. To see the properties for these fields, in the application tree view, inside the Data Dictionary folder, double-click the name of the field.

Fields are defined to have a class ID, which in turn determine the properties the field has. When you add fields, you can either add them with predefined class IDs, or you can create your own class IDs.

Table 14.1 shows the Data Dictionary for the Insurance Claim Processing application:

Table 14.1 Data dictionary for Insurance Claim Processing application

Data Field	Purpose
Customer_Name	Stores the customer name
Customer_Address	Stores the customer address.
Customer_Email	Stores the customer email, so that the email can be sent to the customer whenever a claim is approved, denied or a process is expired.
Customer_PhoneNo	Stores the customer phone number, which can be needed in case you need to contact the customer for clarifications.
Policy_No	Stores the policy number of the customer.
Start_Date	Stores the policy start date.
End_Date	Stores the policy end date.
Policy_Amt	Stores the policy amount
Deductible	Stores the policy deductible amount.
Vehicle_Model	Stores the vehicle model
Vehicle_IDN	Stores the identification number for the vehicle
AppComments	Stores the approver's comments
CalrifyComments	Stores the clarifier's comments.
Claim	Stores the customer's claim
claimid	Stores a randomly generated claim ID
Window_damage	checkbox for predefined claim - window damage
Body_damage	checkbox for predefined claim - body damage
Engine_damage	checkbox for predefined claim - engine damage
Accessories_damage	checkbox for predefined claim - accessories damage

Forms

The following sections describe the forms included in the Insurance Claim application.

Entry.html

Customers enter the Policy Number to begin the Insurance Claim process in this form, as shown in Figure 14.3.

Figure 14.3 Entry.html

The screenshot shows a web form titled "Enter Policy Number". The form contains a welcome message, instructions for claiming insurance, and a text input field for the "Insurance Policy No." with the value "P12365". A "Next" button is located at the bottom right of the form area. Below the form, there is a status bar indicating a connection to the host.

Enter Policy Number

Welcome ! This is the BPM Insurance Claim Process Management System.

To claim your Insurance, we would like you to go through certain steps before we could process your request.

In order to get your claim approved, please provide us with your Insurance Policy No. as on your Policy Documents.

Insurance Policy No. :

Next

Connect: Host onguard contacted. Waiting for reply...

Resubmit.html

This form, shown in Figure 14.4, is displayed whenever the Policy Number, which the customer entered in Entry.htm, is not found in the flat file.

Figure 14.4 Resubmit.html

[Process Map](#)

Re-enter Policy Number

ERROR !!

Policy Number Not Found !!

In order to get your claim approved, please *RE-ENTER* your Insurance Policy No. and your Name as on your Policy.

Insurance Policy No. :

You can add additional comments here that will appear in the Details & Histor

Connect: Host onguard contacted. Waiting for reply...

Details.html

This form, shown in Figure 14.5, displays the customer's personal information and the policy information that was found in the flat file.

Figure 14.5 Details.html

<u>Policy Holder Information</u>	
Your Record has been Located.	
<u>Personal Information</u>	
Customer Name : Mike Sijacic	
Customer Address : 655 North Fair Oaks, Suite 9, Sunnyvale, CA-94086 -	
Customer Phone No. : (408)545-4442	
Customer Email : sijacic@netscape.com	
<u>Policy Details</u>	
Policy Number : P12365	
Policy Start Date : 1-1-1998	Policy End Date : 31-12-1999
Policy Amount : 12000.0	Policy Deductible : 1000.0
Vehicle Model : BMW 535	Vehicle IDN : THE CAR
Connect: Host onguard contacted. Waiting for reply...	

EnterClaim.html

Customers enter the claim in this form, shown in Figure 14.6. It has a predefined claim check boxes and a text area where the customer can enter a small description of the claim.

Figure 14.6 EnterClaim.html

The screenshot shows a web form titled "Enter Claim Details" with a header bar. Below the title, there is a prompt: "Please Enter your Claim Details." To the right, there is a table with four rows, each containing a checkbox and a label. The first row has a checked checkbox and the label "Vehicle Body Damage". The other three rows have unchecked checkboxes and labels "Engine Damage", "Accessories Damage", and "Window Damage". Below this table, there is another prompt: "Please fill in a brief description of the damage." followed by a text area containing the text "Rear Bumper broken." and "Lights have been damaged." At the bottom of the form, there is a footer bar with the text "<help for Claim>".

<u>Enter Claim Details</u>	
Please Enter your Claim Details.	
<input checked="" type="checkbox"/>	Vehicle Body Damage
<input type="checkbox"/>	Engine Damage
<input type="checkbox"/>	Accessories Damage
<input type="checkbox"/>	Window Damage

Please fill in a brief description of the damage.

Rear Bumper broken.
Lights have been damaged.

<help for Claim>

Confirmation.html

This form, shown in Figure 14.7, displays customer details from Details.html and claim details from EnterClaim.html.

Figure 14.7 Confirmation.html

Work Item List	New Process	Search
<p>Before Processing you Claim, please confirm the following :</p>		
<p><u>Personal Information</u></p>		
Customer Name : Mike Sijacic		
Customer Address : 655 North Fair Oaks, Suite 9, Sunnyvale, CA-94086 -		
Customer Phone No. : (408)545-4442		
Customer Email : sijacic@netscape.com		
<p><u>Policy Details</u></p>		
Policy Number : P12365		
Policy Start Date : 1-1-1998	Policy End Date : 31-12-1999	
Policy Amount : 12000.0	Policy Deductible : 1000.0	
Vehicle Model : BMW 535	Vehicle IDN : THE CAR	
<p><u>Claim Details</u></p>		
		<input checked="" type="checkbox"/> Vehicle Body Damage <input type="checkbox"/> Engine Damage <input type="checkbox"/> Accesories Damage <input type="checkbox"/> Window Damage
<p><u>Claim Description</u></p>		
Rear Bumper broken. Lights have been damaged.		
<p>Connect: Host onguard contacted. Waiting for reply...</p>		

Approver.html

This form, shown in Figure 14.8, is same as Confirmation.html with an additional text area for the approver's comments.

Figure 14.8 Approver.html

Work Item List	New Process	Search
<u>Policy Details</u>		
Policy Number : P12365		
Policy Start Date : 1-1-1998	Policy End Date : 31-12-1999	
Policy Amount : 12000.0	Policy Deductible : 1000.0	
Vehicle Model : BMW 535	Vehicle IDN : THE CAR	
<u>Claim Details</u>		
<input checked="" type="checkbox"/> Vehicle Body Damage <input type="checkbox"/> Engine Damage <input type="checkbox"/> Accessories Damage <input type="checkbox"/> Window Damage		
<u>Claim Description</u>		
Rear Bumper broken. Lights have been damaged.		
<u>Comments</u>		
<div style="border: 1px solid black; padding: 5px; min-height: 80px;"> Check the deductible in the Policy. </div>		
<u>Changes in the Policy(if any)</u>		
<help for AppComments>		

Clarify.html

This form, shown in Figure 14.9, is similar to Approve.html but has an additional text area for the clarifier's comments.

Figure 14.9 Clarify.html

Work Item List	New Process	Search
Policy Number : P12365		
Policy Start Date : 1-1-1998	Policy End Date : 31-12-1999	
Policy Amount : 12000.0	Policy Deductible : 1000.0	
Vehicle Model : BMW 535	Vehicle IDN : THE CAR	

Claim Details

<input checked="" type="checkbox"/> Vehicle Body Damage
<input type="checkbox"/> Engine Damage
<input type="checkbox"/> Accesories Damage
<input type="checkbox"/> Window Damage

Claim Description

Rear Bumper broken.
Lights have been damaged.

Approver Comments

Check the deductible in the Policy.

Changes to the Policy (if any)

Deductible changed.

<help for ClarifyComments>

InfCustomer.html

This form, shown in Figure 14.10, is displayed whenever a claim is approved. It shows the customer's name, customer's address, and the policy number.

Figure 14.10 InfCustomer.html

The screenshot shows a web browser window with a title bar that reads "Process Map". The main content area is titled "Inform Customer" in a bold, underlined font. Below the title, the text reads: "Congratulations, Mike Sijacic !!". This is followed by the message: "Your Insurance Claim for Insurance Policy No. : P12365 has been cleared." The next line states: "The BPM Insurance Company will be posting you your check at 655 North Fair Oaks, Suite 9, Sunnyvale, CA-5". Below this is the text: "Thank You for you co-operation and patience. We look forward to serve you better." At the bottom of the main content area, there is a line of italicized text: "(If you have any questions/comments about this process, feel free to call 1-800-BPM-APPS)". Below the main content area, there is a horizontal line, and then a text input field with the placeholder text: "You can add additional comments here that will appear in the Details & His". At the very bottom of the browser window, there is a status bar that reads: "Connect: Host onguard contacted. Waiting for reply..."

SendCheck.html

This form (see Figure 14.11) tells the customer that the claim check was sent.

Figure 14.11 SendCheck.html

[Process Map](#)

Check Sent

System Message

CHECK SENT for Policy No. P12365 in the name of Mike Sijacic at 655 North Fair Oaks, Suit

You can add additional comments here that will appear in the Details & Histor

Connect: Host onguard contacted. Waiting for reply...

ClaimApproved.html

This exit form is displayed whenever a claim is approved and the process is successfully completed.

ClaimDenied.html

This exit form is displayed whenever a claim is denied and the process is successfully completed.

ExpirationExit.html

This exit form is displayed whenever a form has expired and it goes to the exit point.

Script Dictionary

The script dictionary contains the scripts used by the application. They are divided into categories based on the purpose they serve in the application.

Expiration Scripts

The expiration scripts run whenever a form has not been used or accessed for a period of time set by an expiration handler. When the period of time has elapsed without the form being accessed, the expiration handler script is executed.

The activities with expiration scripts in this application are set to expire in five minutes if the form is not accessed. In each activity's property window, the expiration setter is set to five minutes. Each activity has an expiration handler script that is called when the five minutes has passed. The script advances the process to the Expiration exit point.

The following sections show the expiration scripts at the activities where they occur:

Script at Policy Details Activity

```
function exppolicydetails ( )
{
    var pi = getProcessInstance ( );
    var wi = getWorkItem ( );

    // handle expiration
    // move process instance to Expiration Exit point.
    wi.setNodeName ( "Expiration Exit" );
    return true;
}
```

Script at Enter Claim Details Activity

```
function expenterclaim ( )
{
    var pi = getProcessInstance ( );
    var wi = getWorkItem ( );

    // handle expiration
    // move process instance to Expiration Exit point.
```



```

        wi.setNodeName ( "Expiration Exit" );
        return true;
    }

```

Script at Confirmation

```

function expconfirmation ( )
    var pi = getProcessInstance ( );
    var wi = getWorkItem ( );

    // handle expiration
    // move process instance to Expiration Exit point.
    wi.setNodeName ( "Expiration Exit" );
    return true;
}

```

Script at Resubmit Policy No.

```

function expresubmit ( )
    var pi = getProcessInstance ( );
    var wi = getWorkItem ( );

    // handle expiration
    // move process instance to Expiration Exit point.
    wi.setNodeName ( "Expiration Exit" );
    return true;
}

```

Completion Scripts

Completion scripts are executed whenever an activity goes on to the next step. This is particularly helpful whenever you want to do a particular task such as displaying the form or setting some value in a data field as soon as the activity is completed.

In this application, after displaying at the policy details in Details.html, the application displays the EnterClaim.html form directly, without returning the user to their work item list. The completion script displays the form automatically to the user by using a function to redirect the URL.

The following sections show the completion scripts at the activities where they occur:

Completion Script at Policy Details Activity

```
function Enterclaim ( )
{
    var pi = getProcessInstance( );
    var process_id = pi.getInstanceId( );

    url = "http://onguard/cgi-bin/gx.cgi/
AppLogic+wfmySampleApp.npm?eventId=OnDisplayWorkItem&__instanceId="
    + process_id +
    "&__forkId=0&__nodeName=Enter%20Claim%20Details";

    setRedirectionURL(url);
    return true;
}
```

Completion Script at Enter Claim Details Activity

```
function Confirmation ( )
{
    var pi = getProcessInstance( );
    var process_id = pi.getInstanceId( );

    setRedirectionURL("http://onguard/cgi-bin/gx.cgi/
AppLogic+wfmySampleApp.npm?eventId=OnDisplayWorkItem&__instanceId="
    + process_id +
    "&__forkId=0&__nodeName=Confirmation");

    return true;
}
```

Required Files

The following sections show the additional required files for the Insurance Claim application.

The database.xml File

The following is an example of the database.xml file you need in order to run the application. The policy numbers listed in the file are the ones you need to enter in order to process an application.

```
<xml version="1.0" encoding="us-ascii">
    <POLICYSET>
```

```

<POLICY>
  <CUSTOMER_NAME>Mike Sijacic</CUSTOMER_NAME>
  <CUSTOMER_ADDRESS>
    876 North Fair Oaks, Suite 9, Sunnyvale, CA-94086
  </CUSTOMER_ADDRESS>
  <CUSTOMER_PHONE_NUMBER>(408)123-4442
  </CUSTOMER_PHONE_NUMBER>
  <EMAIL> sijacic@netscape.com </EMAIL>
  <POLICY_NUMBER>P12365</POLICY_NUMBER>
  <START_DATE>1-1-1998</START_DATE>
  <END_DATE>31-12-1999</END_DATE>
  <POLICY_AMOUNT>12000</POLICY_AMOUNT>
  <POLICY_DEDUCTIBLE>1000</POLICY_DEDUCTIBLE>
  <VEHICLE_MODEL>BMW 535</VEHICLE_MODEL>
  <VEHICLE_NUMBER>THE CAR</VEHICLE_NUMBER>
</POLICY>
<POLICY>
  <CUSTOMER_NAME>MICHAL CHMIELEWSKI</CUSTOMER_NAME>
  <CUSTOMER_ADDRESS>
    3248 LAKEVIEW APTS, San Jose, CA-94086
  </CUSTOMER_ADDRESS>
  <CUSTOMER_PHONE_NUMBER>(408)123-4372
  </CUSTOMER_PHONE_NUMBER>
  <EMAIL> michal@netscape.com </EMAIL>
  <POLICY_NUMBER>P32345</POLICY_NUMBER>
  <START_DATE>1-1-1998</START_DATE>
  <END_DATE>31-12-1999</END_DATE>
  <POLICY_AMOUNT>12000</POLICY_AMOUNT>
  <POLICY_DEDUCTIBLE>1000</POLICY_DEDUCTIBLE>
  <VEHICLE_MODEL>BMW 747</VEHICLE_MODEL>
  <VEHICLE_NUMBER>seeCAR23</VEHICLE_NUMBER>
</POLICY>
<POLICY>
  <CUSTOMER_NAME>Souvik Das</CUSTOMER_NAME>
  <CUSTOMER_ADDRESS>
    844 Salt Lake, Mountain View, CA-94043
  </CUSTOMER_ADDRESS>
  <CUSTOMER_PHONE_NUMBER>(650)123-2343
  </CUSTOMER_PHONE_NUMBER>
  <EMAIL> souvik@netscape.com </EMAIL>
  <POLICY_NUMBER>P12346</POLICY_NUMBER>
  <START_DATE>1-1-1998</START_DATE>

```

```

<END_DATE>31-12-1999</END_DATE>
<POLICY_AMOUNT>12300</POLICY_AMOUNT>
<POLICY_DEDUCTIBLE>400</POLICY_DEDUCTIBLE>
<VEHICLE_MODEL>HONDA Accord V6 LE</VEHICLE_MODEL>
<VEHICLE_NUMBER>PF12335</VEHICLE_NUMBER>
</POLICY>
<POLICY>
  <CUSTOMER_NAME>Albert TAM</CUSTOMER_NAME>
  <CUSTOMER_ADDRESS>884 Mary Blvd., Mountain View, CA-94043
  </CUSTOMER_ADDRESS>
  <CUSTOMER_PHONE_NUMBER>(650)123-4578
  </CUSTOMER_PHONE_NUMBER>
  <EMAIL> atam@netscape.com </EMAIL>
  <POLICY_NUMBER>P12347</POLICY_NUMBER>
  <START_DATE>1-1-1998</START_DATE>
  <END_DATE>31-12-1999</END_DATE>
  <POLICY_AMOUNT>12000</POLICY_AMOUNT>
  <POLICY_DEDUCTIBLE>1000</POLICY_DEDUCTIBLE>
  <VEHICLE_MODEL>BMW 535 v6</VEHICLE_MODEL>
  <VEHICLE_NUMBER>BPM GROUPIE3</VEHICLE_NUMBER>
</POLICY>
</POLICYSET>

```

The Policy.log File

The following is an example of the log file that is created when you run the application. It contains the ID numbers and claim information of process instances.

```

[Tue Aug 17 16:32:30 PDT 1999]
The Insurance Claim with Claim id : 1715745114 for Policy No. : P12346
was initialized by : Souvik Das
[Tue Aug 17 16:33:27 PDT 1999]
Claim ID : 1715745114 With Claim : ' Broken fuel tank ' was submitted
[Tue Aug 17 16:33:27 PDT 1999]
Claim ID : 1715745114 was APPROVED.

[Tue Aug 17 16:34:04 PDT 1999]
The Insurance Claim with Claim id : 618655790 for Policy No. : P12365
was initialized by : Mike Sijacic
[Tue Aug 17 16:34:52 PDT 1999]
Claim ID : 618655790 With Claim : ' All windows broken ' was denied
REASON : ' null '

```

The Banner Image

The application uses a banner image, shown in Figure 14.12, on all displayed forms.

Figure 14.12 The banner.gif file



The Background Image

The application uses a background image, shown in Figure 14.13, on all displayed forms.

Figure 14.13 The background image is called Gray_Textured1040.gif



Configuring the Insurance Claim Processing Application

Before running the Insurance Claim Processing application, make sure the `database.xml` file resides in the correct location in your environment. This file contains sample policy data. The application retrieves this data for a policy that the user enters.

On Solaris, `database.xml` must reside in the home directory of the user who is running the application server. On Windows NT, you must place `database.xml` in the root of the drive of where Windows NT is installed. For example, if Windows NT is installed on the `E:` drive, then `database.xml` file must reside at `E:\database.xml`.

Note that the `policy.log` file will be created in the same location as `database.xml`.

Custom Activity Code

This section summarizes the three custom activities in the Claim Processing sample application:

- The `LogPerformer` Activity
- The `LogdenialPerformer` Activity
- The `LookupPerformer` Activity

For information on writing and packaging a custom activity, see Chapter 17, “Writing Custom Activities.”

The `LogPerformer` Activity

The `LogPerformer` custom activity writes specific information to a log file. This custom activity is represented by the `LogPerformer.jar` archive, which contains the following files:

```
/LogPerformer.xml  
/com/model/sampleapp/logperformer/Logger.class  
/com/model/sampleapp/logperformer/LogPerformer.class
```

The `LogdenialPerformer` Activity

The `LogdenialPerformer` custom activity writes a claim process denial to a log file. This custom activity is represented by the `LogdenialPerformer.jar` archive, which contains the following files:

```

/LogdenialPerformer.xml
/com/model/sampleapp/logdenialperformer/Logger.class
/com/model/sampleapp/logdenialperformer/LogdenialPerformer.class

```

The LookupPerformer Activity

The LookupPerformer custom activity takes the policy number as input, searches a database of policies, and returns the policy details back to the process. This custom activity is represented by the `LookupPerformer.jar` archive, which contains the following files:

```

/LookupPerformer.xml
/com/model/sampleapp/lookupperformer/Logger.class
/com/model/sampleapp/lookupperformer/LookupPerformer.class
/com/model/sampleapp/lookupperformer/MyParser.class
/com/model/sampleapp/lookupperformer/Policy.class
/com/model/sampleapp/lookupperformer/PolicySet.class

```

Code Walkthrough for LookupPerformer.java

This section describes the steps for creating the `LookupPerformer.java` file. This file contains three main parts:

- Definitions and Packages
- The `init`, `perform`, and `destroy` Methods
- The `GetPolicy` Method

Definitions and Packages

The first part of `LookupPerformer.java` is written as follows:

1. Define a package for your class:

```
package com.model.sampleapp.lookupperformer;
```

2. Import the required Java packages:

```
import java.lang.*;
import java.util.*;

import com.model.sampleapp.lookupperformer.*;
import com.netscape.pm.model.*;
import java.io.*;
```

3. A custom activity must implement the `ISimpleWorkPerformer` interface, so you define the `LookupPerformer` class to do so:

```
public class LookupPerformer implements
    com.netscape.pm.model.ISimpleWorkPerformer
{
```

4. Next, define the variables to use within the custom activity:

```
private String CustomerName ;
private String CustomerAddress ;
private String CustomerPhoneNo ;
private String CustomerEmail ;
private double PolicyAmt ;
private String PolicyNo ;
private String StartDate ;
private String EndDate ;
private double Deductible ;
private String VehicleModel ;
private String VIN ;

private String mFileName ;
```

The `init`, `perform`, and `destroy` Methods

1. The `init` method is used to set up anything you'll need globally within the custom activity. In this example, the method initializes the name and location of the file to use to search for policies.

```
public void init( Hashtable env ) throws Exception {
    mFileName = "database.xml";
}
```

2. The `perform` method does the main work of the custom activity.

```
public void perform( Hashtable input, Hashtable output )
    throws Exception {
```


This method takes data from an input hashtable (which is passed in from the process), does the necessary work, and then puts the data back into an output hashtable (which is passed back to the process).

3. The data hashtable from the process is in the `input` variable. Similarly, the hashtable in which to pass data back to the process is in the `output` variable.

```
// Read the Policy_No attribute from the input hashtable
Object value = input.get ( "Policy_No" ) ;
```

4. Next, you check to see whether the value from the hashtable is in the proper format, a string.

```
if ( value instanceof com.netscape.javascript.NativeJavaObject ) {
    input.put (
        "Policy_No", ((com.netscape.javascript.NativeJavaObject)
            value).unwrap().toString() ) ;
}
String mPolicyNo = (String) input.get( "Policy_No" ) ;
```

5. A call to `GetPolicy` will parse the `database.xml` file and return the details of the policy.

```
Policy ourPolicy = GetPolicy (mPolicyNo) ;
```

6. Check whether the policy number received by the custom activity is blank. If so, return an empty policy number in the output hashtable. You'll check whether the policy number returned within the process is empty and show an error screen there.

```
if ( ourPolicy == null ){
    // throw new Exception("Policy No not found");
    PolicyNo = " ";
    output.put ( "PolicyNo", PolicyNo ) ;
    return;
}
```

7. Next, generate a random number to assign as the claim number.

```
String RanNum = null ;
Random rgen = new Random () ;
RanNum = Integer.toString (rgen.nextInt());

if ( RanNum.startsWith ("-") ) {
```

```

        RanNum = RanNum.substring ( 1 ) ;
    }

```

- Now construct a message line for logging. The message contains the claim number and some of the policy details.

```

String PNo = (String) input.get( "Policy_No" );
StringBuffer templ = new StringBuffer
    ( "The Insurance Claim with Claim id : " );
templ.append ( RanNum )
templ.append ( " for Policy No. : " + PNo);
CustomerName = ourPolicy.getCustomerName ( ) ;
templ.append ( " was initialized by : " + CustomerName );

```

- Open a Logger object that writes the information to a log file.

```

Logger logger = new Logger ();
logger.open();
logger.log ( templ.toString() ) ;
logger.close();

```

- Define variables based on the ourPolicy object, which contains the details of the customer and policy.

```

CustomerName = ourPolicy.getCustomerName ( ) ;
CustomerAddress = ourPolicy.getCustomerAddress ( ) ;
CustomerPhoneNo = ourPolicy.getCustomerPhoneNo ( ) ;
CustomerEmail = ourPolicy.getCustomerEmail ( ) ;
PolicyNo = ourPolicy.getPolicyNo( ) ;
PolicyAmt = ourPolicy.getPolicyAmt( ) ;
StartDate = ourPolicy.getStartDate( ) ;
EndDate = ourPolicy.getEndDate( ) ;
Deductible = ourPolicy.getDeductible( ) ;
VehicleModel = ourPolicy.getVehicleModel( ) ;
VIN = ourPolicy.getVIN( ) ;

```

- Next, populate the output hashtable with the information from the policy.

```

// Set the output hashtable
output.put ( "CustomerName", CustomerName ) ;
output.put ( "CustomerAddress", CustomerAddress ) ;
output.put ( "CustomerPhoneNo", CustomerPhoneNo ) ;
output.put ( "CustomerEmail", CustomerEmail ) ;
output.put ( "PolicyNo", PolicyNo ) ;

```

```

output.put ( "PolicyAmt", Double.toString(PolicyAmt) ) ;
output.put ( "StartDate", StartDate ) ;
output.put ( "EndDate", EndDate ) ;
output.put ( "Deductible", Double.toString(Deductible) ) ;
output.put ( "VehicleModel", VehicleModel ) ;
output.put ( "VIN", VIN ) ;
output.put ( "ClaimId", RanNum );
// End of perform method
}

```

12. The `destroy` method is invoked when the application is unloaded from the application server. It is the opportunity to clean up resources that are used by the server. In this case, there is nothing to do.

```

public void destroy( ){
}

```

The GetPolicy Method

The `GetPolicy` method returns a policy object that contains the information on a given policy. The method itself opens up the `database.xml` file and parses through it to find the appropriate policy. Here is the code for this method:

```

private Policy GetPolicy (String PNO)
{
    try
    {
        String fileName = "database.xml";
        int Pfound = 0;

        int MAX_LENGTH = 50000 ;
        char [] xml = new char [ MAX_LENGTH ] ;

        java.io.File f = new java.io.File ( fileName ) ;
        FileReader fr = new FileReader ( f ) ;
        BufferedReader in = new BufferedReader( fr );

        int count = 0 ;
        while ( (count = in.read ( xml , count , MAX_LENGTH ) ) != -1 ) {
        }

        String xmlStr = new String ( xml ) ;
    }
}

```

```
MyParser parser = new MyParser ( xmlStr ) ;
PolicySet tempPolicySet = parser.parse ( ) ;
int j;

Policy policy = null ;
for ( j=0 ; j < tempPolicySet.GetSize(); j++ )
{
    policy = (Policy ) tempPolicySet.elementAt(j) ;
    if ( policy.getPolicyNo().equals (PNO ) )
    {
        Pfound = 1 ;
        System.out.println("PNO in GET POLICY IS : " + PNO );
        System.out.println( "Policy No Found!!");
        return policy;
    }
}
return null;      // If Policy not found ...

} catch ( Exception ignore ) {
    return null;
}
}
```

Advanced Techniques for Scripting

This chapter describes how to write your own scripts for use with Process Builder.

This chapter has the following sections:

- Introduction
- Getting Information about the Current Process
- Getting Information about Users and their Attributes
- Accessing the Content Store
- Logging Error and Informational Messages
- Verifying Form Input
- Initializing and Shutting Down Applications
- Debugging Hints
- Sample Scripts

Introduction

When writing a PAE script, consider the types of information your script will use. PAE scripts can use the following sources of information:

- Information about the process in progress.

This data is stored in PAE's database, and is available through the `processInstance` object. This information includes data such as the date the process instance was created and the current value of fields such as the `specialDiscount` field.

- Information about the work item in progress.

Information about the current work item is available through the `workItem` object. This information includes data such as the user to whom the work item is assigned.

- Information about users.

This information is stored in the Directory Server, and is available through the `corporateDirectory` object, discussed in “Getting Information about Users and their Attributes”. This data includes information about users, such as the assignee's address or the name of their manager.

- Items stored in the content store.

Scripts can access items stored in the content store by using the `contentStore` object (discussed in “Accessing the Content Store”).

Getting Information about the Current Process

PAE provides two JavaScript objects, `processInstance` and `workItem`, that scripts can use to access information about the process in progress:

- `processInstance` holds information about the process instance in general, such as its creation date and the current values of any data fields. Scripts can use the global function `getProcessInstance()` to get the object. Scripts can then call methods on the `processInstance` to get and set data relevant to the process instance.
- `workItem` holds information about the current work item, such as its assignee and its expiration date. Scripts can use the global function `getWorkItem()` to get the object. Scripts can then call methods on the `workItem` object to get and set data relevant to the work item.

See Appendix A, “JavaScript API Reference” for details on these methods.

Getting and Setting Data Field Values

When a user submits a form for a work item, the value of each data field in the form is put into the corresponding data field in PAE's database. For example, if the user sets the value of the field `documentName` to `Jo Writer` then when the form is submitted the field `documentName` in PAE's database gets the value `"Jo Writer"`. (You could think of the field in the form as being a window to the field in the database.)

There are several ways to verify the data that users enter in the form fields, as discussed in "Verifying Form Input".

Scripts can use the following methods on the `processInstance` object to get and set data field values:

- `getData (fieldName)`

This method returns the value of the named field.

- `setData (fieldName, fieldValue)`

This method sets the value of the named field to the given value. This method is particularly useful for setting values in automation scripts.

The following function is an example of a completion script that gets and sets field values:

```
function setOrderPrice () {
    // Get the process instance.
    var pi = getProcessInstance ();
    // Get the value of the item_price field.
    var price = pi.getData ("item_price");
    // Get the value of the number_ordered field.
    var numOrdered = pi.getData ("number_ordered");
    // Calculate the total price.
    var totalPrice = price * numOrdered;
    // Put the total price in the order_price field.
    pi.setData ("order_price", totalPrice);
    // Successful completion scripts return true.
    return true;
}
```

Getting Data Field Values in Decision Point and Automation Script Transitions

The value of a condition for a decision point or an automated script can be any expression that returns `true` or `false`.

These expressions can use a shorthand notation to refer to field values by simply using the name of the field. For example:

```
item_price < 100
```

is shorthand for

```
getProcessInstance().getData("item_price") < 100
```

This expression evaluates to `true` if the value of the `item_price` field is less than 100, otherwise it evaluates to `false`.

Note that only scripts used in conditions can use this shorthand for accessing data fields.

Getting Information about Users and their Attributes

Scripts can use the global function `getCorporateDirectory()` to get an object that represents the corporate directory in the Directory Server that PAE uses.

The Directory Server contains a database of users and their attributes, such as their distinguished name, their common name, their phone number, their address, their email address, and so on. Given the `corporateDirectory` object, scripts can get information about the users in the corporate directory.

Finding Users and Accessing their Attributes

The `corporateDirectory` object has three methods that allow scripts to search for users in the corporate directory:

- `getUserByCN (userCN)`
- `getUserByDN (userDN)`
- `getUserById (userID)`

These methods all return a JavaScript object that represents the user if they can be found; otherwise the methods return `null`. The object has variables for all the user's attributes. See Appendix A, "JavaScript API Reference" for details.

Scripts can also use the following two methods on the `processInstance` object to get user attributes from the Directory Server:

- `getCreatorUser()` returns a JavaScript object containing the attributes of the user that created the process instance.
- `getRoleUser(roleName)` returns a JavaScript object containing the attributes of the user assigned to the given role.

For example, the following assignment script assigns the work item to the user whose distinguished name is the value of the `peerReviewerDN` attribute of the person who created the process instance.

```
assignToPeerReviewer () {
    var pi = getProcessInstance ();
    var creator = pi.getCreatorUser ();
    // Assignment scripts return an array of distinguished names.
    return new Array (creator.peerReviewerDN);
}
```

Note that this script works only if the `peerReviewerDN` attribute is defined in the Directory Server and is initialized for the creator user.

Modifying User Attributes

The `corporateDirectory` object has the following methods for adding, replacing, or deleting user attribute values in the Directory Server:

- `modifyUserById (userID, attrName, attrValue, operation)`
- `modifyUserByCN (userCN, attrName, attrValue, operation)`
- `modifyUserByDN (userDN, attrName, attrValue, operation)`

These methods specify the user either by ID or common name as appropriate. The `attrName` parameter is the name of the attribute to be modified. If operation is `ADD`, the value `attrValue` is added to any existing values. If operation is `REPLACE`, all existing values are replaced by `attrValue`. If operation is `DELETE`, then `attrValue` is deleted from the attribute. The functions return `true` if the operation is successful, otherwise they return `false`.

For example, the following code puts the value `nikki@company.com` to the `mail` attribute of the user whose common name is `Nikki Beckwell`:

```
var corpdir = getCorporateDirectory();
corpdir.modifyUserByCN ("Nikki Beckwell", "mail", "nikki@company.com",
"REPLACE");
```

Although scripts can modify attribute values, they cannot create attributes that have not already been defined in the Directory Server schema.

Verifying an Array of User DNs

Scripts, particularly assignment scripts, can use the global function `checkUserDNs()` to verify that each element of an array is a valid distinguished name (DN). This script returns `true` if each element of the array is a valid distinguished name, or if the array is empty. It returns `false` if any element is not a valid distinguished name.

The following example script creates an array containing two distinguished names. It checks that each element of the array is a valid distinguished name, and if so, returns the array. Otherwise the script returns an array containing the distinguished name of the creator. It is unlikely that a script would manually build distinguished names in this way, but the point of this script is to illustrate the use of `checkUserDNs()`.

```
assignToJackAndTopper () {
    var topper = "uid=topper, o = airius.com, cn = Topper Kent";
    var jack = "uid=jack, o = airius.com, cn = Jack Alford";
    var assigneeArray = new Array (topper, jack);
    // Check that this array is valid.
    if ((assigneeArray.length != 0) && (checkUserDNs (assigneeArray)))
        return assigneeArray;
    else
```

```

    return new Array (pi.getCreatorDN());
}

```

Adding and Deleting Users

The `corporateDirectory` object has the following methods for adding and deleting users to the corporate directory:

- `deleteUserByCN (userCN)`
- `deleteUserByDN (userDN)`
- `deleteUserById (userID)`
- `addUser (userDN, attributes, objectClasses)`

See Appendix A, “JavaScript API Reference” for details.

Accessing the Content Store

Scripts can use the global function `getContentStore()` to get a `contentStore` connected to the content store for the process.

The following global function is useful for constructing a file name when you want to add an item to the object store:

- `getBaseForFileName()` returns the folder name that the current process instance uses to contain its stored content.

For more information about methods on the content store, see Appendix A, “JavaScript API Reference.”

Example of Accessing a Stored Item

For an example of accessing the content store, consider a process that writes use for submitting documents for review. Before starting the process, the writer must create a file containing a synopsis of the document.

The entry point form contains a File data field named `docDescription`. The writer selects a file containing the document synopsis and uploads it by pressing the File icon in the form. The next activity after the entry point is an automated activity that uses the following automation script. This script gets the document synopsis by getting the content of the file that the writer selected. The script then puts the synopsis into the `docSynopsis` TextArea datafield. The intention here is that the next form in the process displays the synopsis to the users assigned to review the document.

```
function getDocDescription (){
    // Use the getProcessInstance() global function to get a reference
    // to the process instance.
    var pi = getProcessInstance();

    // Retrieve the URL where the file containing the document
    // description is stored in the content store. This was set when
    // the writer selected a File containing a document description.
    var descriptionURL = pi.getData("docDescription");

    // Get a reference to the content store service through the
    // global function getContentStore().
    var contentStore = getContentStore();

    // Get the content of the file.
    var docDescription = contentStore.getContent(descriptionURL);

    // Put the document description in a text field called docSynopsis.
    pi.setData ("docSynopsis", docDescription);
    // Return true to proceed to the next activity.
    return true;
}
```

Note: This script requires that the content store URL defined for the application is a valid URL, such as:

```
http://pm.company.com/pmstore
```

If the content store URL is not defined correctly, the `store()` method will not work.

Storing Files in the Content Store

Each Process Engine uses a series of directories named `part0`, `part1`, `part2` and so on to contain items in the object store. Each process instance has a unique identity, such as `pi0001`. Each process instance uses a `partn` subdirectory to store all its content. For example, all the stored files associated with process instance `pi0001` would be stored in `PAE_root/part0/pi0001`. When `part0` gets full, the Process Engine starts using `part1`, and so on.

When a script needs to save content to the content store, it can use the `getBaseForFileName()` function to get the pathname for the content store directory for the current process instance. The script can then use the `store()` method on the `contentStore` object to save content to a file in the correct content store directory for the current process instance.

For example, the following automation script code creates a path name for the file `myNewFile.html` in the content store, then saves some content to a new file in the content store.

```
function storeSomething () {
    var pi = getProcessInstance();
    // Create a pathname for a new file in the content store.
    var pi = getProcessInstance ();
    var pid = pi.getInstanceId();
    var contentStorePath = getBaseForFileName (pid);
    var newFileName = contentStorePath + "myNewFile.html";

    // Create a string containing some content to be saved to the file.
    var newContent = "Do something that generates a content string here";

    // Store the content in the file myNewFile.html.
    var myStore = getContentStore ();
    myStore.store (newContent, newFileName);
    // Return true to proceed to the next work item
    return true;
}
```

Logging Error and Informational Messages

PAE provides several global functions for providing messages. Scripts can use these functions to log error messages, to add entries to the history log, and to add informational messages for the user when a script fails.

These functions are:

- `logErrorMsg()`
- `logInfoMsg()`
- `logSecurityMsg()`
- `logHistoryMsg()`
- `setErrorMsg()`

As a convenience, PAE provides a duplicate name for each of these functions, using `Message` instead of `Msg` in the name. For example, `logInfoMsg()` and `logInfoMessage()` are equivalent. For more information about these global functions, see Appendix A, “JavaScript API Reference.”

Verifying Form Input

This section has the following subsections:

- Verifying Form Input with Client-Side JavaScript
- Verifying Form Data in Completion Scripts

When you create a form in Process Builder, you place data fields on it. If the data field is in edit mode, it displays the current value, if any, and allows the user to modify the value. If a data field is in view mode, it displays the value of the field but does not allow the user to change it.

When the user submits the form by pressing an action button, the values in the data fields in the form are put in the fields in the process instance’s table in the database. By default, PAE verifies that the data entered in each field is the correct type, and if there are obvious errors, it rejects the work item or entry point and displays a warning dialog box. For example, if the user enters the

string value "whenever" into a data field whose type is `Date`, PAE catches the error and rejects the work item. (Note however that the `Date` verification is not very strict, if the user enters 6666/7777/8 as a `Date`, PAE accepts it.)

There will be times when your form needs to perform additional data verification, such as checking that a number falls within a certain range of numbers. PAE provides two ways to verify that users enter valid data for a field:

- use client-side JavaScript scripts to define `onClick` or `onValueChange` properties for those data fields that have them, or to define `onSubmitForm` scripts.
- use `onCompletion` scripts to perform data type verification.

Using client-side scripts to check data field values allows you to write scripts that give the user a chance to fix the values before the form is submitted.

If the data verification occurs in a completion script, then if the script fails the result is that the form disappears, all the data that the user entered is lost, and the web page displays an error message. The user then has to open the form again, re-enter all their data, and try the submission again.

Verifying Form Input with Client-Side JavaScript

Process Builder provides two ways to write client-side scripts to verify form input:

- using an embedded script that defines an `onSubmitForm` function that gets invoked when the form is submitted.
- using `onValueChange` and `onClick` event handlers for specific form elements.

For every data field represented in a form, the form contains a form element that has exactly the same name as the data field. Client-side scripts can access data field values as they are currently displayed in the form. The scripts do this by accessing form elements that have the same name as the data field.

If the form contains an embedded client-side script that defines an `onSubmitForm` function, that function is invoked when the user presses an action button on the form. You can define an `onSubmitForm` function to check values of form elements and offer the user a chance to enter new values in cases where the current values are invalid. If the `onSubmitForm` function returns `false`, the form is not submitted. See the section “`onSubmitForm` Example” for a coded example of an `onSubmitForm` function.

You can also define client-side scripts to check the value of individual form elements. Some data fields have an `onClick` or `onValueChange` property. The value of this property is a JavaScript expression. These properties correspond to event handlers in the associated form elements. For information about form elements, see the “Forms” chapter in the *HTML Tag Reference* at:

<http://developer.netscape.com/docs/manuals/htmlguid/index.htm>

The `onClick` script is fired when the form element is clicked. For elements whose value is changed by clicking, such as radio buttons, the `onValueChange` script behaves just like an `onClick` script, and is fired whenever the user clicks the form element. For other elements, such as text fields, the `onValueChange` script is fired when the form element loses focus after its value has changed, which usually happens when the user presses the return key or clicks elsewhere in the form. To see exactly which data fields have `onClick` and `onValueChange` scripts, see the properties for the different kinds of data fields in Process Builder.

You can define `onClick`, `onValueChange`, and `onSubmitForm` scripts that display an alert box or a confirm box to warn the user that an invalid value has been entered. Use the `alert()` function to display an alert box, and use the `confirm()` function to display a confirm box.

To obtain information about JavaScript errors, enter the URL `javascript:` in your version of Netscape Communicator.

Event Handler Example

The following `onValueChange` script for a text field checks if the value of the text field is a number. If the value is not a number, it displays a dialog box with a warning and resets the page count value to 100 in case the user ignores the warning. If the value is a number, it does nothing. (Note that in event handlers, such as in client-side scripts, you must use single quotes instead of double quotes.)


```

var pageCountNum = parseInt (value);
if (isNaN (pageCountNum)) {
    alert ('You must enter a number as the page count');
    value = 100;
}

```

Figure 15.1 shows the definition of this `onValueChange` event handler in the Inspector window in Process Builder.

Figure 15.1A definition for an `onValueChange` event handler

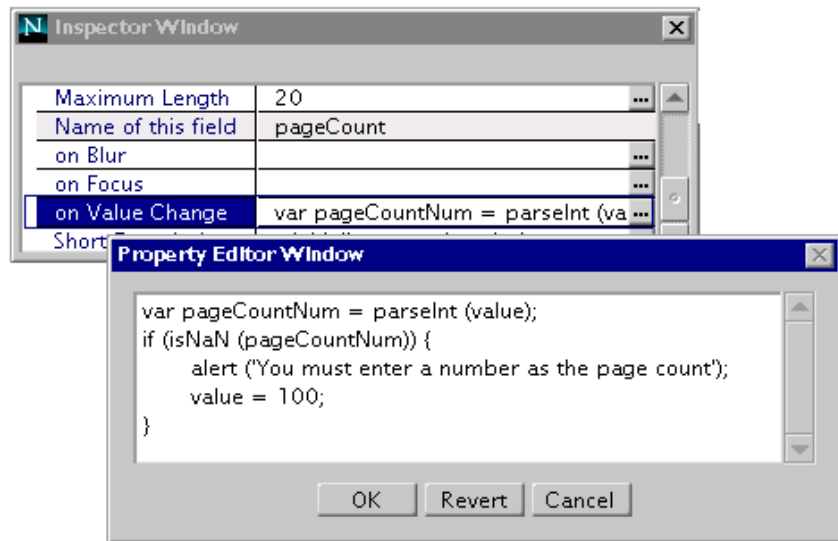
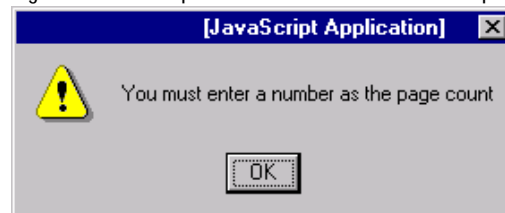


Figure 15.2 shows a sample alert window in Process Express:

Figure 15.2A sample alert window in Process Express



This script can be used directly as the value of the `onValueChange` property of the text field. The script uses `parseInt()` to convert the value of the text field from a string to a number. The `value` argument to the `parseInt` function is the

name of a property of the text field. Since this script is used directly in the text field object, there is no need to say `this.value`, although the script would also work if written as:

```
var pageCountNum = parseInt (this.value);  
// etc...
```

For information about the `parseInt()` method, see Chapter 13, Global Functions in the JavaScript reference at:

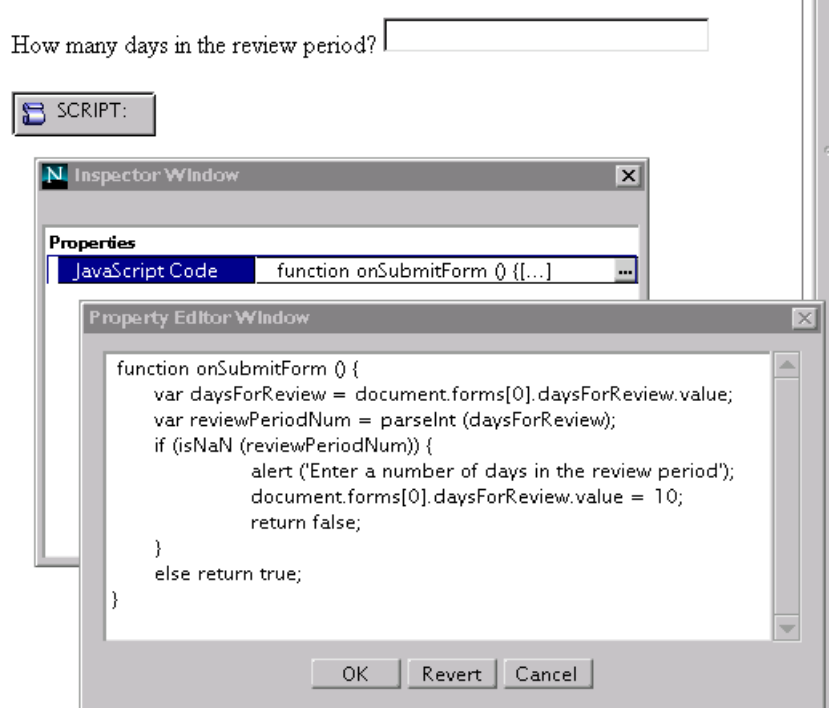
<http://developer.netscape.com/docs/manuals/communicator/jsref/>

onSubmitForm Example

The following example shows an embedded client-side script that defines an `onSubmitForm` function. In this case, the script checks that the value entered in the `daysForReview` form element is a number. If not, the script displays a warning dialog box, sets the value of the `daysForReview` form element to 10, and returns `false` to give the user a chance to change the value and submit the form again.

```
function onSubmitForm () {  
    var daysForReview = document.forms[0].daysForReview.value;  
    var reviewPeriodNum = parseInt (daysForReview);  
    if (isNaN (reviewPeriodNum)) {  
        alert ('Enter a number of days in the review period');  
        document.forms[0].daysForReview.value = 10;  
        return false;  
    }  
    else return true;  
}
```

Figure 15.3 shows an embedded client-side script that defines an `onSubmitForm` function in Process Builder:

Figure 15.3A script that defines an `onSubmitForm` function

Verifying Form Data in Completion Scripts

Some data fields do not have `onClick` or `onValueChange` properties, so you cannot use client side JavaScript scripts to verify their values. Instead, you can use a completion script to verify their data.

When the user presses an action button to submit a form, the values of the data fields on the form are installed into the global `processInstance` object. When the completion script runs, it can access data field values by using the `getData()` method of the global `processInstance` object. However, if the completion script fails, the newly installed values are removed from the process instance and the previous values are re-installed.

If the completion script of an entry point fails, the process instance is not created, and the user must start over again to create the process instance. If the completion script of an activity fails, the user must start over again entering the data for that activity. Whenever a completion script fails, the web page displays an error message and rejects the work item.

Completion scripts can use the `setErrorMessage()` global function provided by PAE to display additional information in the error dialog box, such as adding an explanation as to why the completion script failed.

Initializing and Shutting Down Applications

When defining a process definition, you can write initialisation and shutdown scripts for the application. The initialisation script is invoked when the *application* first starts, and the shutdown script is invoked when the application is shut down. Note that the initialization and shutdown scripts are not invoked each time a process instance starts or finishes, but only when the application starts or finishes. The application lifespan (start to finish) is defined within the scope of each Java engine running within the application server. Therefore, the initialization scripts are invoked on a per-engine basis.

Initialization scripts and shutdown script are not associated with individual process instances, therefore they cannot use `processInstance` or `workItem` objects. They can however use the methods on the `contentStore` object. For example, the initialisation script could use the `store()` method on the `contentStore` object to create a file that can be accessed by all process instances in the application.

You can use the initialization script to perform tasks that need to be done once in the lifetime of the application, such as creating files or initiating connections to be shared by all process instances.

The shutdown script should close any connections that were opened in the initialization script. Other uses for a shutdown script include cleaning up external database tables, sending email to the administrator, releasing other resources, and performing any tasks that need to be done when the application is shut down.

Debugging Hints

To check for syntax errors in your scripts, enter `javascript:` as your URL. In addition, this section provides hints on debugging and troubleshooting scripts. It has the following subsections:

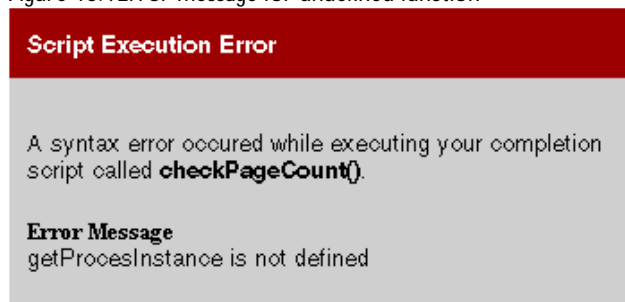
- Undefined Items
- Adding Helpful Messages to Script Failure Dialog Boxes
- Displaying the Progress of a Script
- Testing Expiration Setter and Handler Scripts

Undefined Items

A script may contain undefined items such as functions, variables, or constants. When a script runs in Process Express and encounters an undefined item, PAE displays an error dialog box mentioning the item's name.

For example, Figure 15.4 shows the dialog box that appears if a completion script misspells `getProcessInstance()`.

Figure 15.4 Error message for undefined function



Adding Helpful Messages to Script Failure Dialog Boxes

If a script returns false, PAE displays a dialog box. By default, this dialog box simply says that an error occurred and that you should see the administrator. However, scripts can use the `setErrorMessage()` global function to add an informative explanation to the dialog box. For example:

```
// completion script will return false
{
// Set the error message and return false.
setErrorMessage ("You did not enter a number for the pageCount field");
return false;
}
```

Displaying the Progress of a Script

There are several global functions for printing messages that are very useful for debugging scripts. The `logErrorMsg()` function adds an entry to the error log. The `logInfoMsg()` function adds an entry to the informational log. The `logSecurityMsg()` function adds an entry to the security log.

To view these logs, go to the Process Administrator page (Administrator.apm). For example, if PAE is installed at `pm.company.com`, go to:

```
pm.company.com/Administrator.apm/
```

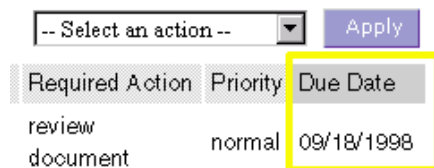
This page displays Process Administrator. Select the Applications tab (if it is not already selected) to see the list of applications. Select View Logs from the pop up menu for the application of interest, then press the Apply button. You may need to enter your user name and password to gain access to PAE and the Enterprise Server.

When the View Logs Server Selector window appears, select the desired cluster, then press the View Log File button. In the View Application Logs window that appears, you can select the Error, Security, or Information log. After making your selection, press the View Log button.

Testing Expiration Setter and Handler Scripts

When Process Express displays work items in the work item list, it shows the expiration date for each work item. To test an expiration setter script, simply deploy the application and test it out. Check the expiration date for the relevant work item in the work item list in Process Express. For example, Figure 15.5 shows the due date in Process Express.

Figure 15.5 Testing a due date



Required Action	Priority	Due Date
review document	normal	09/18/1998

At regular intervals (about once every minute) PAE tests for expired work items by comparing the expiration date and time of every work item to the current time. If the expiration time has passed for any work item, PAE invokes the work item's expiration handler script if it has one.

If your expiration handler re-assigns the work item, you should view the details and history for the process instance to check if the re-assignment worked in test mode. The title of the process does not change in test mode even if the assignee changes.

Sample Scripts

This section gives an example of the following kinds of scripts:

- Assignment Script
- Expiration Setter Script
- Expiration Handler Script
- Completion Script
- Automation Script

Assignment Script

The following assignment script assumes that the process instance has data fields `reviewer1` and `reviewer2` that contain the common name for two reviewers. If one or both of the `reviewer1` and `reviewer2` fields contain common names for valid users, the work item is assigned to the valid named reviewers. Otherwise it is assigned to the process instance creator.

The default value for the `reviewer1` and `reviewer2` fields is an empty string. The script checks that the reviewers exist as users in the corporate directory, and if so, gets their distinguished names (DNs) and puts them in an array. Then just for good measure, the script double-checks that the array contains valid user DNs only, and if it does, returns the array. If the array does not contain valid DNs, the script assigns the work item to the creator of the process instance.

```
function assignToNamedReviewers ()
{
    // Get the process instance.
    var pi = getProcessInstance ();
    var corp = getCorporateDirectory ();

    // Create an array to contain the assignees.
    var assigneeArray = new Array ();

    // Get the common names of reviewer1 and reviewer2.
    // The default value of these fields is an empty string.
    var reviewer1 = pi.getData ("reviewer1");
    var reviewer2 = pi.getData ("reviewer2");

    // If reviewer1 has a name, find them in the corporate directory.
    if (reviewer1 != "") {
        var reviewer1CN = corp.getUserByCN (reviewer1);
        // If we have a user for reviewer1, get their DN
        // and add it to the assignee array.
        if (reviewer1CN != null) {
            var reviewer1DN = reviewer1CN.dn;
            assigneeArray.push (reviewer1DN);
        }
    }

    // Do the same thing for reviewer2.
    if (reviewer2 != "") {
```



```

var reviewer2CN = corp.getUserByCN (reviewer2);
if (reviewer2CN != null) {
    var reviewer2DN = reviewer2CN.dn;
    assigneeArray.push (reviewer2DN);
}
}
// If the assignee array is not empty and each element is a valid dn
// return the assignee array otherwise
// return an array of the dn of creator of the process instance.

if ((assigneeArray.length != 0) && (checkUserDNs (assigneeArray))) {
    return assigneeArray;
}
else {
    return new Array (pi.getCreatorDN());
}
}

```

Expiration Setter Script

This expiration setter script gets the value of the `pageCount` field and uses it to determine whether reviewers need a long or short review period. If the page count is greater than 100, the review period is long (14 days), otherwise the page count is short (7 days).

```

function setEndOfReviewPeriod(){
    // Get the process instance.
    var pi = getProcessInstance();

    // Get the page count.
    var pageCount = pi.getData ("pageCount");

    // The review period is based on the page count.
    // For pages < 100, expire in 7 days.
    // For pages > 100, expire in 14 days.
    var reviewPeriod;
    if (pageCount > 100) {
        reviewPeriod = 14;
    }
    else {
        reviewPeriod = 7;
    }
}

```

```

var now = new Date();
var nowTime = now.getTime(); //UNIX epoch time
var expTime = nowTime + (reviewPeriod * 24 * 60 * 60 * 1000);
return new Date(expTime);
}

```

Expiration Handler Script

This expiration handler script reassigns the work item to the creator. In this particular case, the task of the work item is to review a document, whose name is stored in the docname data field.

```

function reviewPeriodExpired(){

    // Get the process instance and the work item.
    var pi = getProcessInstance ();
    var wi = getWorkItem();

    // Re-assign the work item to the creator
    var creatorArray = new Array (pi.getCreatorDN());
    wi.assignTo(creatorArray);

    // Get the document name.
    var docname = pi.getData("docname");
    // getCurrentActivityCN() is a work item method that returns
    // the name of the work item.
    var activityName = wi.getCurrentActivityCN();

    // Return true to indicate successful expiration.
    return true;
}

```

Completion Script

The following completion script checks that the value of the pageCount data field is a number between 1 and 3000 inclusive. If the pageCount is too high, too low, or is not a number, the completion script adds an informative error message to the error dialog box and returns false.

```

function checkPageCount(){
    var pi = getProcessInstance ();
    var pageCount = pi.getData("pageCount");

    // The standard JavaScript function parseInt gets an integer
    // from a string if it has one, otherwise returns 0.
    // For example, parseInt("33hello") returns 33.
    // If the first character cannot be converted to a number,
    // parseInt returns "NaN".
    var pageCountNum = parseInt(pageCount);

    // The standard JavaScript function isNaN
    // tests if a value is not a number.
    if ((isNaN (pageCountNum)) ||
        (pageCountNum < 1) || (pageCountNum > 3000)) {
        setErrorMessage ("You entered " + pageCount +
            " as the page count. This value is invalid. " +
            "Please enter the page count as a number " +
            "between 1 and 3000 inclusive.");
        return false;
    }
    else return true;
}

```

Automation Script

This script updates a link to a newly submitted document in a file that lists all documents available for review.

This script could be used as an automation script in a process that enables writers to submit documents for review. In the entry point form, the writer uses a File attachment data field called `docFileName` to attach the document. Many different writers can use the application to submit documents. The file `docList.html` contains an active link for every document that has been submitted with this application. The file `docList.html` lives in the top level of the content store so that all process instances in the application can access it.

Two files are involved in the document list. The file `docList.html` is an HTML-formatted list of the documents with an active link for each document. The file `showList.html` contains opening HTML tags, the list of documents

(which is the same as `docList.html`), and closing HTML tags. This script uses two files so that it can add the closing HTML tags after adding the information for the newly submitted document.

```
function updateDocReviewWebPage(){
// Get the process instance.
var pi = getProcessInstance();

// docList.html and showList.html are stored at the
// top level in the content store.

// Get the root URL for the content store.
var myStore = getContentStore();
var storePath = myStore.getRootURL();

// Get the full pathnames to docList.html and showList.html.
var docListPath = storePath + "docList.html";
var showListPath = storePath + "showList.html";

// Get the contents of the doc list.
var docList = myStore.getContent(docListPath);

// Get the pathname to the doc that was submitted for review.
var docURL = pi.getData("docFileName");

// Get info about the newly submitted document.
var docname = pi.getData("docname");
var writer = pi.getData("writername");
var writerComments = pi.getData("writerComment");

// Add a link to the newly submitted doc in the doc list.
docList += "<A HREF=" + docURL + ">";
docList += "<H2><FONT COLOR='#55CCAA'> " + docname + " </H2></FONT></A>";
docList += "<P>Author: " + writer + "</P>";
docList += "<P>Author comments: </P>";
docList += "<BLOCKQUOTE>" + writerComments + "</BLOCKQUOTE><HR>";

// Store the doclist back into docList.html
myStore.store (docList, docListPath);

// Create the final content, which contains the heading tags,
// the doc list, and the closing tags.
```

```

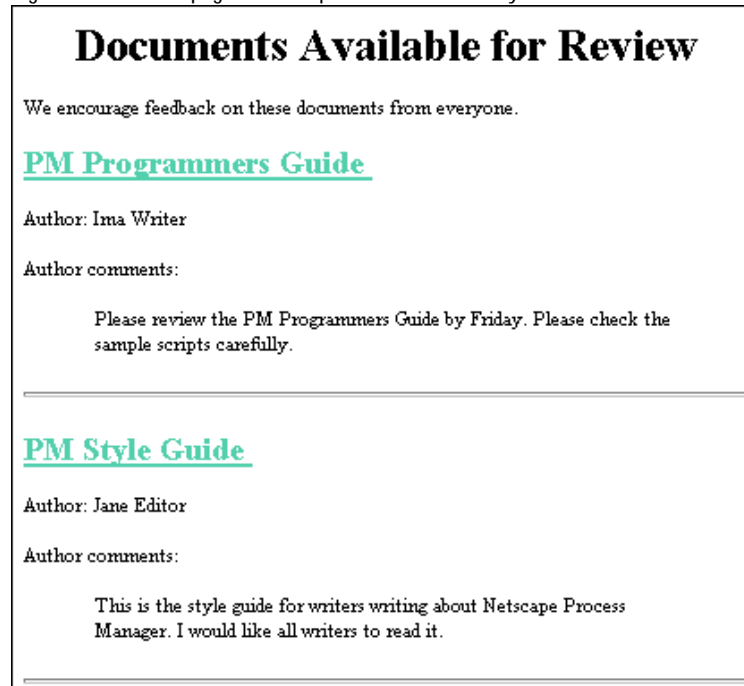
var finalContent = "<HTML><HEAD><TITLE>Documents available for review" +
    "</TITLE></HEAD>";
finalContent += "<BODY><CENTER><H1>Documents Available for Review" +
    "</H1></CENTER>" +
    "<P>We encourage feedback on these documents from everyone.</P>" +
docList + "</BODY></HTML>";
myStore.store (finalContent, showListPath);

return true;
}

```

Figure 15.6 illustrates the `showList.html` web page. This page is updated each time a document is submitted for review.

Figure 15.6A web page that is updated automatically



Scripting with EJB Components

This chapter describes how to call an Enterprise JavaBean (EJB) from JavaScript. You should be familiar with EJBs to understand the information in this chapter.

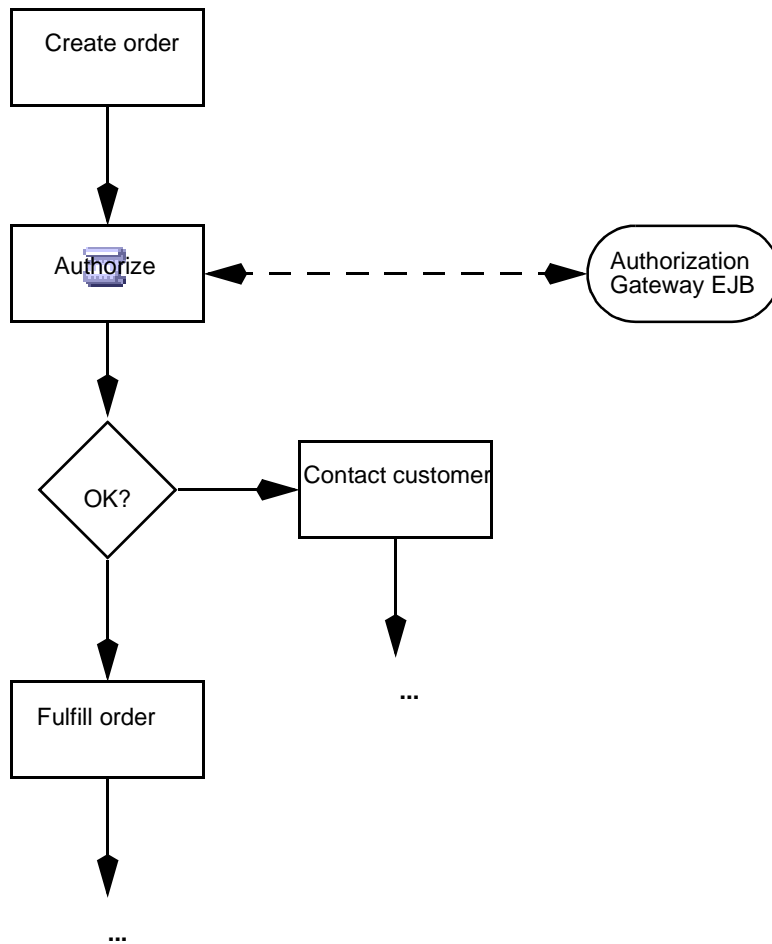
This chapter describes the following topics:

- Calling EJB Components from JavaScript
- A Sample Script
- Handling Exceptions
- Data Conversion Issues

Calling EJB Components from JavaScript

Your JavaScript process automation script can create an instance of an EJB and call the EJB's methods to perform business logic. For example, an order-entry application might create an instance of an EJB that represents a credit card authorization gateway. The application would call methods to check the credit card number, authorize the transaction, and create a charge on the account, as shown in Figure 16.1:

Figure 16.1 Programming logic for calling an EJB component



A Sample Script

To use an EJB, the calling JavaScript must perform the following actions:

- Look up the bean using the home interface's JNDI name
- Create an instance of the bean
- Call the bean's methods to perform tasks supported by the bean

The following JavaScript function shows an example of these actions:

```
function billCC ()
{
    var pi = getProcessInstance ();
    var wi = getWorkItem();
    pi.setData("ccError",null);
    var ccNumber = pi.getData("ccNumber");
    // Get bean home, create it, and call the check method on it.
    var home = ejbLookup("Netscape/CreditCardServer");
    var ccs = home.create ( pi.getData("ccType") );

    var result = ccs.check ( ccNumber );
    if (result.equals
    ( Packages.com.netscape.pm.sample.ICreditCardServer.CARD_OK )
    == false)
    {
        pi.setData("ccError", result);
        return true;
    }

    result = ccs.authorize ( ccNumber, 100, 25 );
    if (result.startsWith (
    Packages.com.netscape.pm.sample.ICreditCardServer.CREDIT_LIMIT_EXCEEDED)
    || result.startsWith
    Packages.com.netscape.pm.sample.ICreditCardServer.CREDIT_DENIED)
    )
    {
        pi.setData("ccError", result);
        return true;
    }
    // Charge the card the amount shown.
    result = ccs.charge ( ccNumber, result );
    if (result.startsWith
    ( Packages.com.netscape.pm.sample.ICreditCardServer.OPERATION_OK )
    == false)
    {
        pi.setData("ccError", result);
        return true;
    }
}
```

```
    }  
    pi.setData("ccAuthcode", result);  
    return true;  
}
```

You call the `ejbLookup()` function to create a reference to the EJB. This method uses the bean's JNDI name to determine the EJB to use. In this example, the name is `Netscape/CreditCardServer`. You can examine the bean's `BeanHomeName` property to determine the JNDI name.

You call the bean's factory method, which is defined in the bean's home interface class, to create the instance of the bean. In this example, the method is `create()`. You can examine the bean's `HomeInterfaceClassName` property to determine the home interface class.

You can then call the bean's methods, which are defined in the bean's remote interface class, to perform specific tasks. In this example, the JavaScript calls the bean's `check()`, `authorize()`, and `charge()` methods. You can examine the bean's `RemoteInterfaceClassName` property to determine the remote interface class.

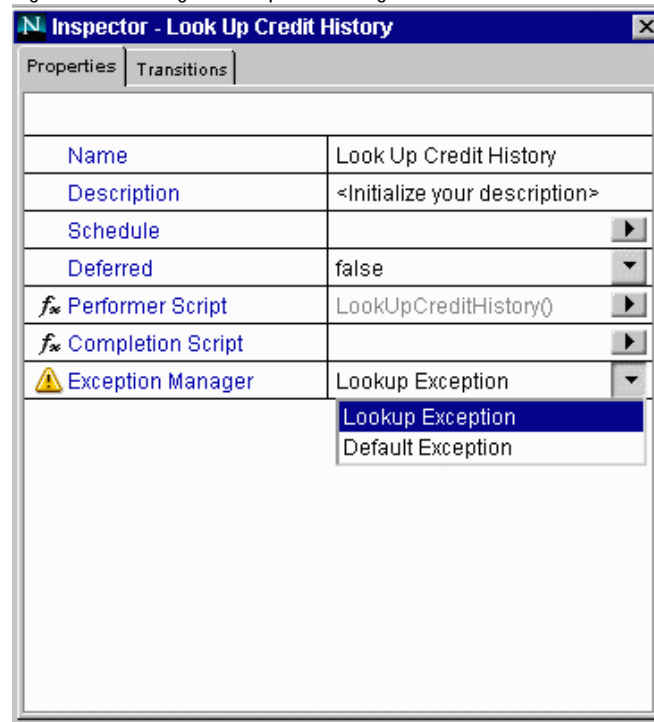
Handling Exceptions

Java exceptions cannot be handled in JavaScript. You can implement your bean so that the script receives error information without needing to deal with an exception, as in the following example:

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

If an exception is thrown, the process instance moves to the exception manager that you specify in the Inspector window, as shown in Figure 16.2:

Figure 16.2 Setting an exception manager in Process Builder



Data Conversion Issues

PAE uses LiveConnect's rules to convert between Java and JavaScript data types. When you call a Java method from JavaScript, JavaScript does not convert the data type of the data until it is manipulated by the script. For example, the `result` variable in the script contains a Java string data type after executing the Java EJB `check()` method in the following statement:

```
var result = ccs.check ( ccNumber );
```

You must compare the variable's contents using Java methods; comparing with JavaScript functions will not work as expected. The following statement shows the comparison using Java:

```
if (result.equals  
( Packages.com.netscape.pm.sample.ICreditCardServer.CARD_OK )  
    == false) { ... }
```

To access Java data, you must qualify the package name with the `Packages` keyword unless the Java data is from the `java` package. JavaScript recognizes the `java` package without qualification. You can also create an instance of an object by specifying the `new` operator, as in the following examples:

```
var aResult = new java.lang.String;  
var CARD_OK = new  
    Packages.com.netscape.pm.sample.ICreditCardServer.CARD_OK
```

Note Conversion between Java and JavaScript is implicit and occurs when the value is manipulated as a JavaScript data type. For example, if `result` contains a Java string data type, the following statement causes it to be converted to a JavaScript string data type: `result = result + ""`;

Writing Custom Activities

This chapter describes how to write and use custom activities. The sections in this document are:

- Introduction
- Implementing ISimpleWorkPerformer
- Writing the XML Description File
- Packaging a Custom Activity
- Adding a Custom Activity to the Process Map
- Working with a Custom Activity
- Implementation Tips

Introduction

PAE lets you create custom activities as Java classes and bring them into your process definitions.

Custom activities are useful when you want to do more than can easily be done in an automation script, such as when the programming logic or data resides outside of PAE. For example, you might build a custom activity to interface with external applications and databases. Custom activities might also run local applications and then interact with mail gateways or FAX servers.

Comparison to Automated Activities

Custom activities are similar to automated activities. In both cases:

- You place them on the process map by dragging and dropping them from the Palette.
- They can have completion scripts.
- They are triggered as soon as the process instance reaches the activity, unless the activity is deferred. A deferred activity is triggered at its specified date and time.

Automated and custom activities have one main difference: an automated activity is carried out by an automation script, whereas a custom activity is carried out by a Java class.

Usage Overview

Creating and using a custom activity involves the following major steps:

1. Write and compile a Java class that implements the `ISimpleWorkPerformer` interface.
2. Define an XML description file for the activity.
3. Package the Java class and the XML description file as a zip or jar file.
4. Bring the custom activity into an application.

Implementing ISimpleWorkPerformer

The first step in creating a custom activity is to write a Java class that implements `ISimpleWorkPerformer`, an interface in the package `com.netscape.pm.model`. The Java class must reside in the server class path. Note also that the `ISimpleWorkPerformer` class you create will be stateless.

`ISimpleWorkPerformer` defines a custom activity that:

1. gets data field values as input
2. performs some task
3. sets data field values as output

Other than getting and setting data field values, your `ISimpleWorkPerformer` class has no access to information on the work item or the process instance.

This section describes the following topics:

- Methods of `ISimpleWorkPerformer`
- Sample Java Class

Methods of ISimpleWorkPerformer

`ISimpleWorkPerformer` has three methods:

- The `init()` method is called when the application starts.
- The `perform()` method is called each time the custom activity is executed. This method must be thread-safe.
- The `destroy()` method is called when the application is unloaded or removed.

The `init()` method

```
public void init (Hashtable environment) throws Exception
```

The `init()` method performs initialization tasks that the custom activity requires when the application starts. For example, use `init()` to set up database connections that are shared by all instances of the activity, or use `init()` to define variables that are constant across all instances of the activity.

The `init()` method does *not* execute each time a custom activity is created in a process instance. Instead, this method is called only once—when the application starts.

As its input argument, `init()` takes a hashtable of environment variables. A hashtable is a `java.util.Hashtable` object that contains a series of parameter-value pairs. The parameters in the environment hashtable are defined in the `ENVIRONMENT` section of an XML description file.

A process designer sets the values of the hashtable parameters while creating the process map.

For example, suppose a `Language` parameter is defined in the environment hashtable of a custom activity. In Process Builder, the `Language` parameter would appear as a property for the custom activity (you would open the Inspector window and view the Properties tab).

In your Java class, define the `init()` method to perform the desired initialization tasks. Then, to obtain the value of a parameter in the environment hashtable, call the `get()` method on the environment hashtable. The `get()` method returns either the value of the parameter, or `null` if the parameter doesn't exist.

The `perform()` method

```
public void perform (Hashtable in, Hashtable out) throws Exception
```

The `perform()` method does whatever tasks must be done for the activity. This method takes two `java.util.Hashtable` arguments. The input hashtable contains values taken from data fields, and the output hashtable contains values to put into data fields.

The parameters in the input and output hashtables are defined in the `INPUT` and `OUTPUT` sections, respectively, of an XML description file.

The Input Hashtable

To obtain the value of a parameter in the input hashtable, call the `get()` method on the input hashtable. The `get()` method returns either the value of the parameter, or `null` if the parameter doesn't exist. Note that the `get()` method returns a Java object, so you must cast this object to the object class type that your custom activity is expecting. For example:

```
String sizeOrder = (String) input.get("order");
```

The Output Hashtable

To set data field values, the `perform()` method must put values into the output hashtable by calling `put()` on the output hashtable. When the `perform()` method finishes executing, you then assign the values to the corresponding data fields.

The `destroy()` method

```
public void destroy()
```

The `destroy()` method is called when the application that uses the custom activity is unloaded or removed. Typically, you use the `destroy()` method to clean up resources that were used by the `init()` method.

Sample Java Class

The following code samples are from `HelloWorldPerformer.java`, the class that implements the `HelloWorld` custom activity. `HelloWorld` is included in PAE as a sample custom activity, so you can view the source code directly.

`HelloWorld` constructs a welcome message in either French or English. The message value is derived from two things: the value of the `customerName` data field in the process instance, and the `Language` property of the `HelloWorld` activity instance. The `HelloWorld` activity puts the welcome message in the `greeting` data field.

Creating HelloWorldPerformer.java

Using your favorite Java editor and compiler, create and compile a Java class that implements the `ISimpleWorkPerformer` interface. The compiled class file must reside in the server class path. When you use Process Builder to add a custom activity, PAE automatically places the custom activity's class file in the server's class path.

Note Don't define any constructors in classes implementing `ISimpleWorkPerformer`, because PAE does not use them. A Java exception will be thrown. Defining a class without any constructors is the same as defining one with just a default constructor.

Here are the steps for creating `HelloWorldPerformer.java`:

1. Define a package for your class:

```
package com.netscape.pm.sample;
```

2. Import the required standard Java packages:

```
import java.lang.*;
import java.util.*;
```

3. Define the class `HelloWorldPerformer` to implement `com.netscape.pm.model.ISimpleWorkPerformer`, as follows:

```
public class HelloWorldPerformer
    implements com.netscape.pm.model.ISimpleWorkPerformer
{
```

4. Define two variables to hold the English and French parts of the greeting. Define another variable to hold the complete greeting when it has been derived (such as "Bonjour Nikki.")

```
// Greeting Messages
public static final String GREETING_FRENCH = "Bonjour";
public static final String GREETING_ENGLISH = "Hello";

// Holds the greeting message once the language has been specified
String mGreeting;
```

5. Define the `init()` method to get the value of the Language environment variable and to set the language-specific part of the greeting. In addition, throw an exception if the language is not provided, or if the language is neither English nor French. For example:

```
/**
```

```

* The HelloWorld custom activity knows to generate both French
* and English greetings. The Language argument defines which
* language should be used.
*/
public void init( Hashtable env ) throws Exception
{
    String lang = (String) env.get( "language" );
    if( lang == null )
    {
        throw new Exception( "-- language not defined." );
    }
    else if( lang.equalsIgnoreCase("French") )
    {
        mGreeting = GREETING_FRENCH;
    }
    else if( lang.equalsIgnoreCase("English") )
    {
        mGreeting = GREETING_ENGLISH;
    }
    else
    {
        throw new Exception( "-- Unknown language:"+ lang +
            ". We currently support English or French--" );
    }
}
}

```

Later, you will set the exact value of the Language environment. You'll do this in Process Builder, when you set up the custom activity in a process definition.

6. Define the `perform()` method to construct a welcome message consisting of the language-specific part of the greeting and the user's name, for example "Hello Billy." The value of the `userName` parameter is derived later—from a data field in a process instance that uses the custom activity.

- Use the `get()` method on the input parameter to get the value of an input parameter.

```

/**
* Reads the userName element of the input hashtable,
* generates greetings, and sets the Greeting element of out.
*/
public void perform( Hashtable input, Hashtable output )

```

```

        throws Exception
    {
        // Read the userName attribute from the input hashtable
        String userName = (String) input.get( "userName" );

        if( userName == null )
        {
            throw new Exception( "userName has not been initialized!" );
        }

        // Generate greetings
        String msg = mGreeting + " " +userName;

```

- Use the `put()` method on the output parameter to set the value of an output parameter.

```

        // Put the greeting into the welcomeMsg parameter of
        // the output hashtable.
        output.put( "welcomeMessage" , msg );

```

7. Finally, define the `destroy()` method, which is invoked when the application is unloaded from the application server. In this case, the method does nothing because no resource cleanup is needed.

```

        public void destroy( )
        {
        }
    // End of class
}

```

8. Compile `HelloWorldPerformer.java` to get a class file, `HelloWorldPerformer.class`.

Writing the XML Description File

After you write and compile the Java class that implements `ISimpleWorkPerformer`, your next step is to define an XML description file for the class. This XML file specifies the environment, input, and output parameters

that the class uses. In addition, the XML file specifies some optional design parameters. Design parameters control the custom activity's appearance in Process Builder.

This section describes the following topics:

- File Format
- Sample XML Description File

File Format

The XML description file starts with a tag indicating the XML version, such as:

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

The body of the description is contained between an opening `<WORKPERFORMER>` tag and a closing `</WORKPERFORMER>` tag. Within the `WORKPERFORMER` section you define four sections, as summarized in Table 17.1.

Table 17.1 Sections within the `WORKPERFORMER` Tags

XML Section	What this section describes
ENVIRONMENT	Environment hashtable used by <code>init()</code> method.
INPUT	Input hashtable used by <code>perform()</code> method.
OUTPUT	Output hashtable used by <code>perform()</code> method.
DESIGN	Appearance of custom activity icons in Process Builder.

Here is the structural overview of an XML description file:

```
<?XML version = "1.0" ?>
<WORKPERFORMER >
  <ENVIRONMENT>
    <PARAMETER> ... </PARAMETER> ...
  </ENVIRONMENT>
  <INPUT>
    <PARAMETER> ... </PARAMETER> ...
  </INPUT>
```

```
<OUTPUT>
  <PARAMETER> ... </PARAMETER> ...
</OUTPUT>

<DESIGN>
  <PARAMETER> ... </PARAMETER> ...
</DESIGN>

</WORKPERFORMER>
```

WORKPERFORMER Tag

The `<WORKPERFORMER>` tag has four attributes: `TYPE`, `NAME`, `CLASS_ID`, and `VERSION`.

- `TYPE` is the full package name for the Java class for this type of activity. For a simple custom activity, `TYPE` is always this:

```
com.netscape.pm.model.ISimpleWorkPerformer
```

- `NAME` is the name of the custom activity (which is the same as the name of the XML description file and the jar file that contains the custom activity). This name is not currently used anywhere.
- `CLASS_ID` is the full package name for the Java class that implements the custom activity.
- `VERSION` is the version of the custom activity. `VERSION` is currently unused, but you could use it to keep version information about the description file.

Here is a sample `<WORKPERFORMER>` tag:

```
<WORKPERFORMER
  TYPE="com.netscape.pm.model.ISimpleWorkPerformer"
  NAME="HelloWorld"
  CLASS_ID="com.netscape.pm.sample.HelloWorldPerformer"
  VERSION="1.1">
```

ENVIRONMENT Section

The `<ENVIRONMENT>` tag defines environment parameters that are constant within all instances of the custom activity. For example, suppose that in an application named `HelloWorld`, you set the value of the `Language` environment parameter to `French`. Then, the value is always `French` in every process instance of that application.

The `ENVIRONMENT` section contains embedded `<PARAMETER>` tags. Each `<PARAMETER>` tag describes a parameter in the environment hashtable—the argument used by the `init()` method. The `<ENVIRONMENT>` tag has a corresponding closing `</ENVIRONMENT>` tag, and each `<PARAMETER>` tag has a closing `</PARAMETER>` tag.

When you add the custom activity to the process map in Process Builder, each parameter in the `<ENVIRONMENT>` tag will appear as a field in the Inspector Window.

Here's a sample `ENVIRONMENT` section:

```
<ENVIRONMENT>
  <PARAMETER NAME="Language">"French"</PARAMETER>
</ENVIRONMENT>
```

Warning Parameter values (such as “French” in the example above) are actually JavaScript expressions, so you can supply the value as a string, integer, or function. However, be sure to quote any string expression. Note that `French` (without quotes) and `"French"` (with quotes) mean different things.

For details on the syntax of the `<PARAMETER>` tag, see the section “PARAMETER Tag”.

INPUT Section

The `<INPUT>` tag contains embedded `<PARAMETER>` tags. Each `<PARAMETER>` tag describes a parameter in the input hashtable, the input argument of the `perform()` method. The `<INPUT>` tag has a corresponding closing `</INPUT>` tag, and each `<PARAMETER>` tag has a closing `</PARAMETER>` tag.

To set the value of the parameter to the value of a data field in the process instance, embed a call to `getData()` in the `<PARAMETER>` tag. For example, the following code sets the value of the `userName` parameter in the input hashtable to the value of the `customerName` data field in the process instance.

```
<INPUT>
  <PARAMETER
    NAME="userName"
    DISPLAYNAME="User Name"
    TYPE="java.lang.String"
    DESCRIPTION="Last Name">
    getData("customerName")
  </PARAMETER>
</INPUT>
```

For details on the syntax of the `<PARAMETER>` tag, see the section “PARAMETER Tag”.

The corresponding code in your Java class file uses the `perform()` method to get the value of the `userName` parameter. Within the `perform()` method, you call the `get()` method. Here is a code fragment:

```
public void perform( Hashtable input, Hashtable output )
    throws Exception
{
    // Read the userName attribute from the input hashtable
    String userName = (String) input.get( "userName" );

    if( userName == null )
    {
        throw new Exception( "userName has not been initialized!" );
    }

    // Generate greetings
    String msg = mGreeting + " " +userName;
```

OUTPUT Section

The `<OUTPUT>` tag contains embedded `<PARAMETER>` tags. Each `<PARAMETER>` tag describes a parameter in the output hashtable, the output argument of the `perform()` method. The `<OUTPUT>` tag has a corresponding closing `</OUTPUT>` tag, and each `<PARAMETER>` tag has a closing `</PARAMETER>` tag.

The output hashtable contains parameters whose values will be automatically installed in data fields in the process instance. For each parameter, embed a call to `mapTo()` to indicate which data field in the process instance is to receive the value of the parameter.

For example, the following code specifies that when the `perform()` method has finished executing, the value of the `welcomeMsg` parameter in the output hashtable is automatically installed in the `greeting` data field in the process instance.

```
<OUTPUT>
  <PARAMETER
    NAME="welcomeMsg"
    DISPLAYNAME="Welcome Message"
    TYPE="java.lang.String"
    DESCRIPTION="Greeting for the user">
```



```

    mapTo( "greeting" )
  </PARAMETER>
</OUTPUT>

```

For details on the syntax of the `<PARAMETER>` tag, see the section “PARAMETER Tag”.

The corresponding code in your Java class file uses the `perform()` method to put a value in the `welcomeMsg` parameter of the output hashtable. Within the `perform()` method, call the `put()` method:

```
output.put( "welcomeMessage" , msg );
```

PARAMETER Tag

The `<PARAMETER>` tag has the attributes as summarized in Table 17.2. When you define parameters within the DESIGN section of the XML description file, only the NAME and DESCRIPTION attributes apply. However, within the ENVIRONMENT, INPUT, or OUTPUT sections, all of the attributes apply.

Table 17.2 PARAMETER tag attributes

Attribute	Meaning
NAME	Name of the parameter.
DESCRIPTION	The text for the tool tip (also called bubble help) that appears when you place the mouse over the item in Process Builder.
TYPE	The Java object class of the parameter. This attribute is optional. The value can be given as a complete class name, such as <code>java.lang.String</code> or <code>com.netscape.pm.ShoppingCart</code> .
VALUESET	A comma-delimited list of possible values for this parameter. These values appear as a pop up menu in the Inspector Window. This attribute is optional.
EDITOR	The type of editor window to use. For example, use this attribute to set a Browse button, text area, drop down list, dialog box. This attribute is optional.
EDITABLE	A boolean that determines whether the parameter value can be edited in the Inspector Window. The default is true. This attribute is optional.

DESIGN Section

The `<DESIGN>` tag contains embedded `<PARAMETER>` tags. Each `<PARAMETER>` tag describes a parameter in the output hashtable, the output argument of the `perform()` method. The `<DESIGN>` tag has a corresponding closing `</DESIGN>` tag, and each `<PARAMETER>` tag has a closing `</PARAMETER>` tag.

Use the DESIGN section to define the custom activity's user interface within Process Builder. In the DESIGN section, the `<PARAMETER>` tag accepts two attributes: NAME and DESCRIPTION.

By setting the NAME attribute, you define a particular aspect of the custom activity's user interface. Table 17.3 summarizes the available values for the NAME attribute:

Table 17.3 NAME attributes for the DESIGN parameter

NAME Attribute	Meaning
Icon	The image file to use for the icon in the custom palette.
Label	A text label that appears under the icon.
BubbleHelp	The text for the tool tip that appears when the mouse pointer is over the icon.
HelpUrl	The URL for the online help for this custom activity, accessible from a right-click.
MapIcon	The image file to use for the icon in the process map. In typical usage, this is the same as Icon.
SelectedMapIcon	The image file to use for the icon in the process map, when the activity is selected.
TreeViewIcon	The file to use for a small image that represents the activity in the Application Tree View.

Sample XML Description File

Define a file called `HelloWorld.xml` as shown below. Things to note are:

- This file specifies `userName` as a parameter in the input hash table. However, the value of this parameter is obtained from the `customerName` data field in the process instance.

- Similarly, the file specifies `welcomeMsg` as a parameter in the output hashtable, and maps its value back into the `greeting` data field in the process instance.

Here is the entire code for the `HelloWorld.xml` description file:

```
<?XML version = "1.0" ?>
<WORKPERFORMER
  TYPE="com.netscape.pm.model.ISimpleWorkPerformer"
  NAME="HelloWorld"
  CLASS_ID="com.netscape.pm.sample>HelloWorldPerformer"
  VERSION="1.1">
<ENVIRONMENT>
  <PARAMETER
    NAME="Language"
    VALUESET="'English','French'"
    TYPE="java.lang.String">
    'English'
  </PARAMETER>
</ENVIRONMENT>
<INPUT>
  <PARAMETER
    NAME="userName"
    DISPLAYNAME="User Name"
    TYPE="java.lang.String"
    DESCRIPTION="Last Name">
    getData("customerName")
  </PARAMETER>
</INPUT>
<OUTPUT>
  <PARAMETER
    NAME="welcomeMsg"
    DISPLAYNAME="Welcome Message"
    TYPE="java.lang.String"
    DESCRIPTION="Greeting for the user">
    mapTo("greeting")
  </PARAMETER>
</OUTPUT>
<DESIGN>
  <PARAMETER
    NAME="Icon"
    DESCRIPTION="A 32x32 icon that is placed on the palette">
    drap_uk2.gif
```

```

</PARAMETER>
<PARAMETER
    NAME="Label"
    DESCRIPTION="The DISPLAYNAME used for this palette element.">
Hello World
</PARAMETER>
<PARAMETER
    NAME="BubbleHelp"
    DESCRIPTION="Bubble help for the palette element">
HelloWorld - A simple work performer Custom Activity.
</PARAMETER>
<PARAMETER
    NAME="HelpURL"
    DESCRIPTION="URL explaining this palette element">
http://people.netscape.com/michal/
</PARAMETER>
<PARAMETER
    NAME="MapIcon"
    DESCRIPTION="Icon for the process map (48x48)">
drap_uk2.gif
</PARAMETER>
<PARAMETER
    NAME="SelectedMapIcon"
    DESCRIPTION="Icon for the process map (48x48)">
drap_fr2.gif
</PARAMETER>
<PARAMETER
    NAME="TreeViewIcon"
    DESCRIPTION="Icon for the tree view (48x48)">
mailer_tree_view.gif
</PARAMETER>
</DESIGN>
</WORKPERFORMER>

```

Packaging a Custom Activity

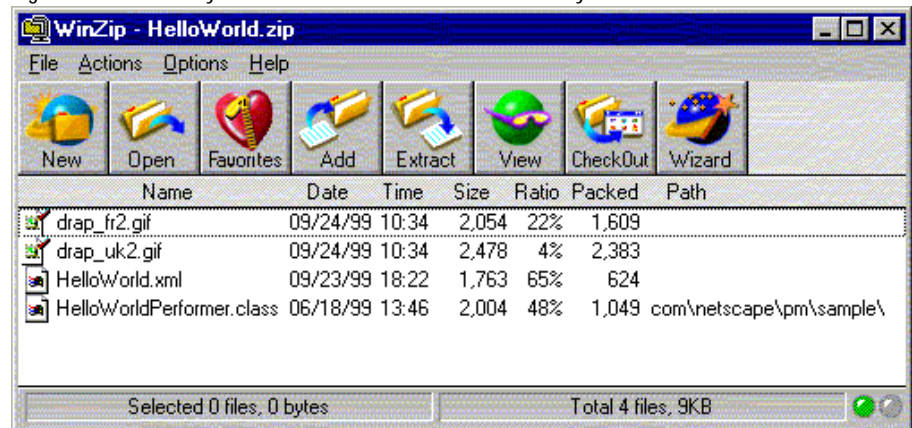
After you create the Java class file and the XML description file, the next step is to package the custom activity. A custom activity consists of the following files:

- One or more Java classes. At least one of these classes must implement `ISimpleWorkPerformer`.
- An XML description file.
- Optional image files to use as icons in Process Builder.

Create a zip or jar archive that contains these files. The archive must have the same root name as the XML file. For example, if the XML file is `HelloWorld.xml`, then name the zip file `HelloWorld.zip`.

As you create the archive, check that the directory structure reflects the package structure of the class. For example, the `HelloWorldPerformer` class is in the package `com.netscape.pm.sample`. Therefore, the class file must be in the directory `com/netscape/pm/sample`, as shown in Figure 17.1. The `HelloWorld.xml` file must be at the top level.

Figure 17.1 Directory structure for the HelloWorld activity



Note the two image files, `drap_fr2.gif` and `drap_uk2.gif`. These images will be used by Process Builder in the process map. The images, shown in Figure 17.2, will correspond to the selected state of the Language property, either French or English.

Figure 17.2 Image files in the HelloWorld activity



Adding a Custom Activity to the Process Map

There are two ways to add a custom activity to the process map:

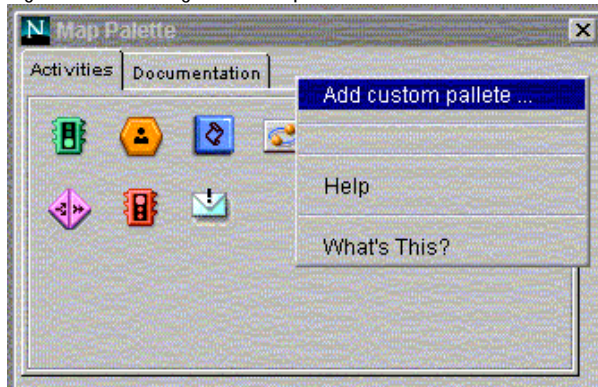
- In one case you create a custom palette. This approach is useful if you intend to use a custom activity often, either within a single application or across several applications.
- In the other case, you don't create a custom palette, and you simply use the Custom Activity icon provided with Process Builder. This approach might be better if you rarely use custom activities, and you don't want to create a custom palette for them.

Adding a Custom Activity from a Custom Palette

To use a custom activity from a custom palette, do the following:

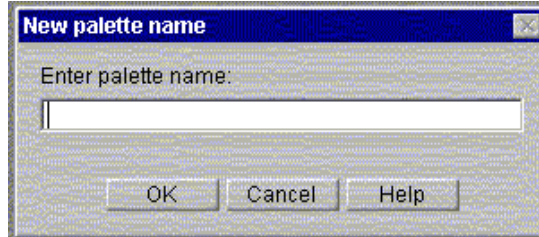
1. In the palette, right-click the area below the title bar, and choose “Add custom palette,” as shown in Figure 17.3. This adds a new tab to the palette.

Figure 17.3 Adding a custom palette



2. In the “New palette name” dialog box (shown in Figure 17.4), type the label for the new tab. For example, enter “HelloWorld”.

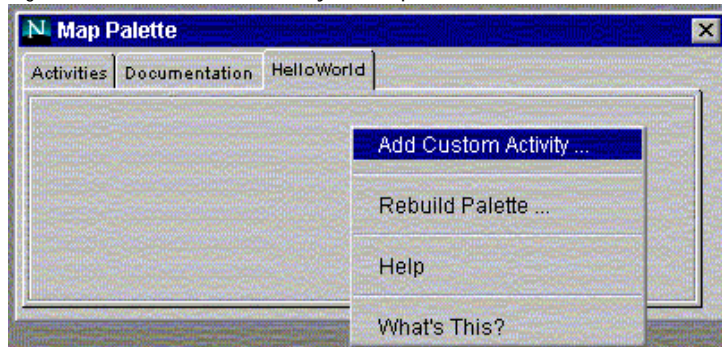
Figure 17.4 Enter a name for the new palette



A new tab is added to the palette.

3. Click your new custom tab to make it active. Note that the area contains no icons.
4. Right-click in the empty area under the tabs, and select “Add Custom Activity ...”. See Figure 17.5.

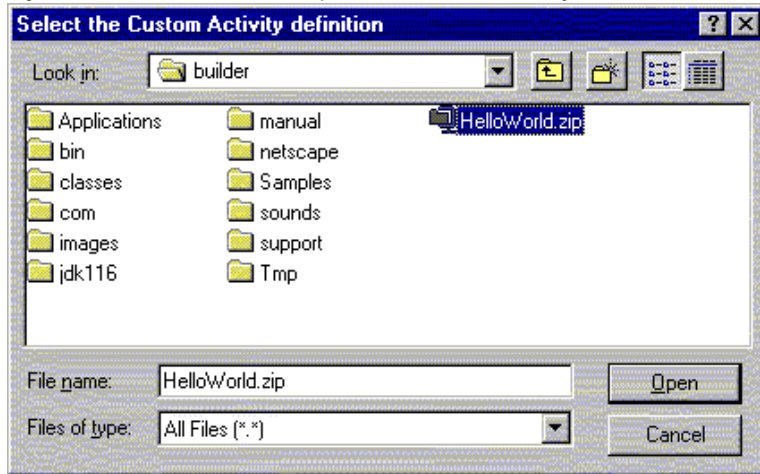
Figure 17.5 Add a custom activity to the palette



A file selection window appears.

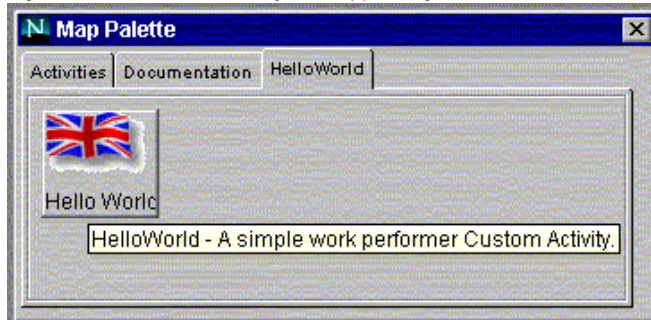
5. Using the file selection window, locate the archive file that represents the custom activity, and select the file. For example, Figure 17.6 show the selection of `HelloWorld.zip`:

Figure 17.6 Select the file that represents a custom activity



The custom activity is added to your new palette. For example, as shown in Figure 17.7, the HelloWorld activity appears on the palette like this:

Figure 17.7A custom activity icon appearing on the HelloWorld custom palette



Note that the custom activity's appearance in Process Builder is controlled by the DESIGN section of the XML file. In the HelloWorld tab pictured above, you see the effects of setting the Icon, Label, and BubbleHelp parameters in the DESIGN section.

6. To add the activity to your application, drag the icon from the custom palette to the process map.

Adding a Custom Activity without Using a Custom Palette

If you don't have a custom palette or don't want to create one, you can add a custom activity as follows:


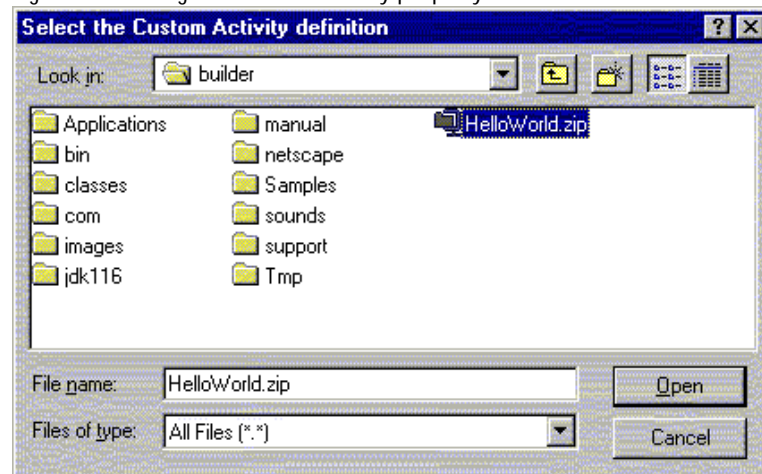
1. In the palette, drag the Custom Activity icon  to the process map.
2. Select the custom activity and open the Inspector window.
3. On the Properties tab of the Inspector, locate the property named Custom Activity.
4. Click the Browse button to bring up a file selection window, and locate the zip or jar file that represents the custom activity. An example is shown in Figure 17.8.

Figure 17.8 Setting the Custom Activity property

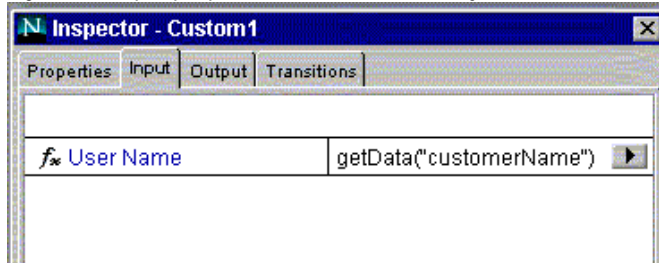


5. Click Open to associate the selected file with the Custom Activity icon. The Custom Activity icon now has the characteristics defined by the file.

Working with a Custom Activity

After you place a custom activity on the process map, you can view or set its properties in the Inspector window. For example, Figure 17.9 shows the Inspector window's Input tab for HelloWorld.

Figure 17.9 Input properties for a custom activity



The Input tab shows the parameter names in the input hashtable, and shows how the value for each parameter is derived. In this case, the value for the input parameter `userName` is derived by getting the value of the `customerName` datafield.

The INPUT section of the XML description file determines the appearance of the Input tab in the Inspector window. For example, note that the `userName` parameter displays as "User Name," which was specified through the `DISPLAYNAME` attribute in the XML file.

Similarly, the Output tab shows the parameter names in the output hashtable, and shows how the value for each parameter is mapped back into the process instance. In this case, the value for the output parameter `welcomeMsg` is put in the `greeting` data field.

As you design the process, be sure to add the data fields that are used by the custom activity. For example, the HelloWorld activity uses two Textfields: `greeting` and `customerName`.

Implementation Tips

This section describes some of the design tips you should consider as you create and implement a custom activity.

Avoid Instance Data

Custom activities, like custom fields, are stateless entities. There can be only one instance of a custom activity per Java Virtual Machine. The custom activity must be multithreaded. This means the activity must safely handle calls from concurrent requests.

As a result, it's recommended that you avoid using instance data in the class that implements a custom activity, particularly if the `perform()` method is likely to change this data. If you can't avoid using instance data, be sure to synchronize the data. With unsynchronized data, a variable set during one request might not exist for the next request.

Use Consistent Data Types

Watch for is consistent data typing. Make sure that the data types you specify in the XML file are consistent with the corresponding values you pass to the input and output hashtables. Although PAE performs some basic data matching for you, inconsistent data is likely to generate an error.

Avoid Non-default Constructors

In classes that implement `ISimpleWorkPerformer`, avoid defining non-default constructors (meaning constructors with non-zero arguments). Otherwise, you may encounter problems during dynamic loading. The problem may arise because PAE dynamically loads the class that implements your custom activity. In other words, PAE has no prior awareness of non-default constructors and therefore cannot call them.

When to Use a Custom Activity

Custom activities are useful when you want to integrate an existing legacy process into a PAE process through a well-defined interface. For example, use a custom activity in a PAE process that exchanges data with external resources such as a CORBA server, a CICS system, or the business logic in an EJB component.

By contrast, custom activities are not a good solution if you must represent a complex data structure from an external source. For example, to represent result sets or other data types from Oracle databases or SAP R/3 systems, you are better off using a custom field. Reserve custom activities for situations where data can be easily parsed and stored (either directly in a data field or in the content store).

Writing Custom Fields

This chapter describes how to write custom fields for use in Process Manager applications.

This chapter includes the following sections:

- Introduction
- Defining Field Properties in a JSB File
- Writing the Java Classes
- Packaging a Custom Field
- Adding a Custom Field to an Application
- Method Reference

Introduction

A data field contains information relevant to a process instance, such as the maximum value of the budget or the name of a document. Process Builder offers a set of predefined data-field classes, such as `Date` and `Textfield`.

Why Use a Custom Field?

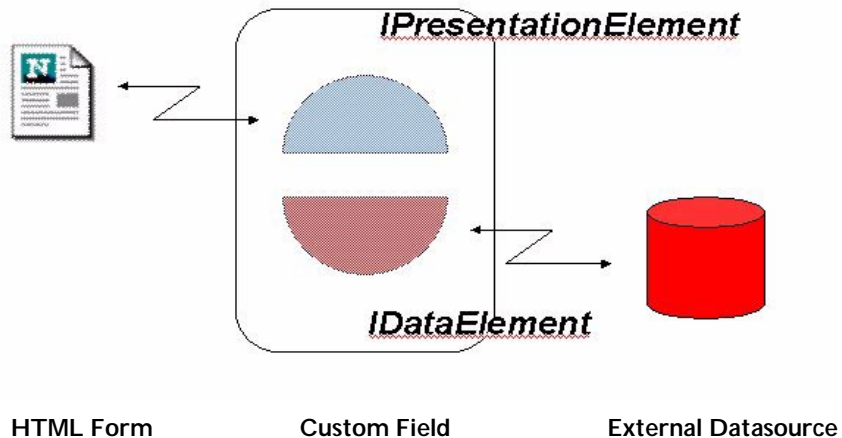
Sometimes you need to create a new class of data field, called a custom field. A custom field is a data field that provides the following additional features:

- support for data types that are more complex than the data types available with built-in fields.
- a convenient way of representing multi-dimensional values, or other high-level data objects, in a process. For example, custom fields can represent a “shopping cart,” an account, or a service order.
- a way of accessing data objects that are stored in resources external to Process Manager, such as PeopleSoft or CICS.

Functional View of a Custom Field

The functional model of a custom field consists of two parts: a presentation component and a data component. Figure 18.1 shows the functional view of a custom field, and its relationship to an HTML form and an external datasource.

Figure 18.1 Functional view of a custom field



The presentation component is represented by the `IPresentationElement` interface. This interface defines the way a custom field will appear in an HTML form. For example, a custom field may appear as a select list, a check box, or a group of radio buttons. `IPresentationElement` also handles reading from and writing to the HTML form.

The data component is represented by the `IDataElement` interface. This interface handles the external storage of the custom field's data. By using a component-based model for custom fields, it will be possible to represent the `IDataElement` object as an entity bean in future releases of PAE.

Steps for Creating a Custom Field

The major steps for creating a custom field are as follows:

- Define the field properties that will be visible in Process Builder. Field properties are defined in a JavaScript bean (JSB) file. For details, see “Defining Field Properties in a JSB File”.
- Write the Java classes that implement the custom field. At a minimum, you must implement two interfaces, `IDataElement` and `IPresentationElement`. For details, see “Writing the Java Classes”.
- Package the JSB and Java classes into a zip or jar archive. For details, see “Packaging a Custom Field”.
- In Process Builder, insert a data field and add the archive file as a new class. For details, see “Adding a Custom Field to an Application”.

Defining Field Properties in a JSB File

The first step in implementing a custom field is to write a JSB file. This file defines the custom's fields properties that will be set at design time, through Process Builder.

In Process Builder, these properties are visible through the field's Inspector window. For each property shown in the Inspector window, a corresponding property is defined in the JSB file.

To create a JSB file for a new custom field class, you can copy an existing JSB file and modify it to suit your needs. For example, you can copy the JSB files for Process Builder's predefined data fields, or you can copy a template JSB file. These files are located in the following path of your Process Builder installation:

```
builder\com\netscape\workflow\fields
```

Warning: Do not modify the original JSB files for predefined data fields. If you do, the data fields may no longer work.

The JSB file and the custom field class must have the same name. For example, a custom field class named `ShoppingCartField.class` must have a JSB file named `ShoppingCartField.jsb`.

A JSB file has the following general structure:

```
<JSB>
  <JSB_DESCRIPTOR ...>
  <JSB_PROPERTY ...>
  <JSB_PROPERTY ...>
  ...
</JSB>
```

The file is surrounded by an opening `<JSB>` tag and a closing `</JSB>` tag. The other two tags are described in the following sections:

- JSB_DESCRIPTOR Tag
- JSB_PROPERTY Tag

JSB_DESCRIPTOR Tag

After the `<JSB>` tag is a `<JSB_DESCRIPTOR>` tag, which specifies the name, display name, and a short description of the data field class.

For example, `ShoppingCartField.jsb` uses the following `<JSB_DESCRIPTOR>` tag:

```
<JSB_DESCRIPTOR
  NAME="com.netscape.pm.sample.ShoppingCartField"
  DISPLAYNAME="Shopping Cart Field"
  SHORTDESCRIPTION="Shopping Cart Field">
```

The `NAME` attribute is the full path name for the data field class, using a dot (.) as the directory separator.

The `DISPLAYNAME` attribute is the name that Process Builder uses for the field, such as the field's name in the Data Dictionary.

The `SHORTDESCRIPTION` attribute is a brief description of the field.

JSB_PROPERTY Tag

After the `<JSB_DESCRIPTOR>` tag comes a series of `<JSB_PROPERTY>` tags, one for each property that appears in the Inspector window. For example, here is one section of `<JSB_PROPERTY>` tags for the `ShoppingCartField.jsb` file:

```
/**
 * Properties related to this specific format of field
 */

<JSB_PROPERTY NAME="dbtype"
    TYPE="string"
    DISPLAYNAME="DB Type"
    SHORTDESCRIPTION="DB Type"
    DEFAULTVALUE="ORACLE"
    VALUESET="ORACLE,SYBASE">

<JSB_PROPERTY NAME="dbidentifier"
    TYPE="string"
    DISPLAYNAME="DB Identifier"
    SHORTDESCRIPTION="DB Identifier">

<JSB_PROPERTY NAME="dbuser"
    TYPE="string"
    DISPLAYNAME="DB User"
    SHORTDESCRIPTION="DB User">

<JSB_PROPERTY NAME="dbpassword"
    TYPE="string"
    DISPLAYNAME="DB Password"
    SHORTDESCRIPTION="DB Password">
```

The `JSB_PROPERTY` attributes and required property names are described in the next two sections.

JSB_PROPERTY Attributes

The attributes for the `JSB_PROPERTY` tag are shown in Table 18.1:

Table 18.1 Attributes for the `JSB_PROPERTY` Tag

Attribute Name	Purpose
NAME	The name of the property.
DISPLAYNAME	The display name for this property, as it appears in the Inspector window.
SHORTDESCRIPTION	A short description of the property.
DEFAULTVALUE	The default value of the property. This attribute is optional.
VALUESET	A comma-delimited list of the possible values for this property. These values appear as a pop up menu on the property in the Inspector window. This attribute is optional.
TYPE	The type of the datafield column in the application table in the database.
ISDESIGNTIMEREADONLY	<p>When specified, this attribute indicates that the property cannot be changed in Process Builder. This attribute is optional. By default, a property value can be changed any time.</p> <p>This attribute does not have an attribute=value specification. You simply give the value, for example:</p> <pre><JSB_PROPERTY NAME="myname" ISDESIGNTIMEREADONLY></pre>
ISEXPERT	<p>When specified, this attribute indicates that the property can be changed in Process Builder while the application is in design mode. This attribute is optional. By default, a property value can be changed any time.</p> <p>This attribute does not have an attribute=value specification. You simply give the value, for example:</p> <pre><JSB_PROPERTY NAME="myname" ISEXPERT></pre>

Required Data Field Properties

Each data field must have the properties listed in Table 18.2:

Table 18.2 Standard Data Field Properties

Property Name	Default Display Name	Purpose
cn	Name of this field	The common name of the data field instance. (Note this is not the name of the data field class.) The name is set when you create the data field in Process Builder.
description	Short Description	A description of the data field.
prettyname	Display Name	The field's display name which is the name that Process Builder uses for the field.
help	Help Message	A help message for the field.
fieldclassid	Field Class ID	This is the package name of the data field class. This is used to ensure that each data field type is unique. This value uses the same convention as the Java naming convention for packages. For example, if <code>ShoppingCartField</code> is stored in <code>\com\netscape\pm\sample</code> , then its <code>fieldclassid</code> is: <code>com.netscape.pm.sample.ShoppingCartField</code>
fieldtype	Data Type	The datatype that the field uses when it is stored in the PAE database. The value must be ENTITY.

In addition to these required properties, each data field has properties that are specific to itself. For example, a `Textfield` has properties for size and length; a radio button data field has a property for options; an applet data field has properties for a class id and parameter; and so on.

When you define the properties for a custom field, consider the purpose of the field. For example, if the custom field must access an external database, you may want to define connection properties. These properties might include the database type (ORACLE, SYBASE), a username and password, or a connection string.

Not all properties you define in a JSB file will necessarily be used. It depends on how your Java class interprets these properties. For example, the JSB file could contain a `color` property that is totally ignored in the Java class. In this case, no matter what color the designer specifies for their field, it will have no effect.

Writing the Java Classes

To write the Java classes for a custom field, perform the following steps:

1. Consider the Design Issues
2. Implement `IPresentationElement` and `IDataElement`

Consider the Design Issues

There are several design issues to bear in mind as you write your custom field classes. This section describes the following design issues:

- Consider Your Data and Data Sources
- Design a Data Manager
- Design a Thread-safe Class
- Use an Entity Key

Consider Your Data and Data Sources

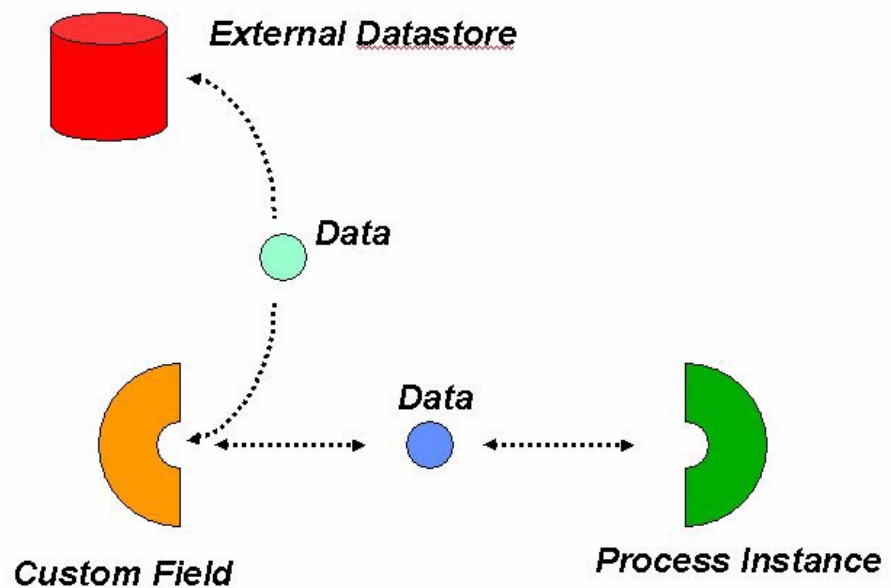
To write your Java classes, you must know something about the data these classes will work with. Consider the following questions:

- What data types do you want the custom field to accept? For example, in what format will the data be? This could well depend on where the data is coming from.
- What data sources will the custom field be required to access? For example, will the custom field access a PeopleSoft application? an SAP R/3 application? a relational database?

Design a Data Manager

A custom field does not represent the data that you are trying to model. Instead, the custom field acts merely as the manager of that data. This concept is shown in Figure 18.2.

Figure 18.2A custom field acts as a data manager



The data can take the form of any Java object. This data is passed from the process instance to the custom field when it is time to store the data. The custom field knows how to store this data into the external repository. Similarly, when the custom field needs to load the data, the field will retrieve

the data from the external repository and pass it to the process instance. The important idea is that the process instance is stateful, and the custom fields specify only the logic to manage the data, not the data itself.

As an example, suppose you have a “shopping cart” custom field, and the data to manage is a set of items stored in a relational database system. Let’s assume that the items are stored as rows in a database table.

To display the shopping cart’s contents, the custom field must retrieve the items from the database. To do so, the custom field will “wrap” the rows with a Java object, thereby providing a well-defined API. As a result, the rest of your process can access the items.

Java Code

Here is the Java code to implement the shopping cart items:

```
public class ItemSet
{
    Hashtable mItems = new Hashtable();

    public void addItem( Item item ) {
        mItems.put( item.getItemId(), item ); }

    public Item getItem( String itemId ) {
        return mItems.get( itemId ); }

    public Enumeration items( ) {
        return mItems.elements(); }
}

public class Item
{
    String mItemId;
    int    mQuantity = 0;
    float  mPrice = 0.0f;

    public Item( String itemId, int quantity, float price )
    {
        mItemId = itemId;
        mQuantity = quantity;
        mPrice = price;
    }

    public String getItemId( ) { return mItemId; }
}
```

```

    public int getQuantity( ) { return mQuantity; }
    public float getPrice( ) { return mPrice; }
}

```

JavaScript Code

The data stored in the process instance will be an instance of `ItemSet`. To retrieve information about each item in the set, you can write an automation script. This script will access the shopping cart custom field by using `getData()`, as shown here:

```

function automationScript( )
{
    var pi = getProcessInstance();
    var items = pi.getData( "shopping_cart" );

    for( var e = items.elements(); e.hasMoreElements(); )
    {
        // we have a reference to an Item object
        var item = e.nextElement();
        var itemId = item.getItemId();
    }

    return true;
}

```

When it is time to store the set of items, the custom field will retrieve the current set from the process instance. The field will then translate the items from their Java object representation to a set of SQL rows.

Design a Thread-safe Class

Custom fields, like custom activities, are stateless entities. There can be only one instance of a custom field per Java Virtual Machine. The custom field must be multithreaded. This means the field must safely handle calls from concurrent requests. As a result, it's recommended that you avoid using instance data in the class that implements a custom field. (But it's safe to have instance variables that store *read-only* configuration information.) If you can't avoid using instance data, be sure to synchronize the data. With unsynchronized data, a variable set during one request might not exist for the next request.

Use an Entity Key

When a custom field loads data from an external data source, the custom field relies on an entity key to identify the data it is looking for. This entity key is stored with the process instance, and the key serves as a handle to the external data. Contrast this with built-in fields. They store the field *value* with the process instance, and there is no need to access external sources.

To work with entity keys, use the following two methods on the `IProcessInstance` interface:

- `getEntityKey(fieldName)`
This call returns the entity key for the custom field whose name is `fieldName`. The entity key is available once the process instance has been loaded. When calling this method, you must convert the retrieved string value into the particular object type you require.
- `setEntityKey(fieldName, key)`
Defines `key` as the entity key for the custom field whose name is `fieldName`. You must call `setEntityKey()` before you store your data. Otherwise, you might not have a handle back to your data. When setting the entity key for a custom field, the key can be any arbitrary object, but the key is stored as a string.

As an example, suppose you have a shopping cart custom field whose entity key is called `basket_id`. This ID number is a unique identifier, distinguishing one basket in the shopping cart table from another basket. If you wanted to use the process instance ID as the basket ID, you would use the following code:

```
long basketID = pi.getInstanceId();
```

Implement IPresentationElement and IDataElement

`IDataElement` has two methods: `display()` and `update()`. `IPresentationElement` has four methods: `create()`, `store()`, `load()`, and `archive()`. You must implement these methods in the Java class for your custom field.

By implementing the methods on `IPresentationElement` and `IDataElement`, you define how your custom field will manage its data objects. However, there are additional methods to implement so that PAE treats your custom field just like any other data field. PAE provides these method implementations for you, through the `BasicCustomField` class.

This class also provides empty stubs for the `IPresentationElement` and `IDataElement` methods that you must fill in.

Therefore, to write your custom field class, use `BasicCustomField` as your base class. For example:

```
import com.netscape.pm.model.BasicCustomField;

public class myCustomField
    extends BasicCustomField
{
    ...
}
```

For implementation details on `IDataElement`, `IPresentationElement`, and `BasicCustomField`, see the section “Method Reference” on page 408.

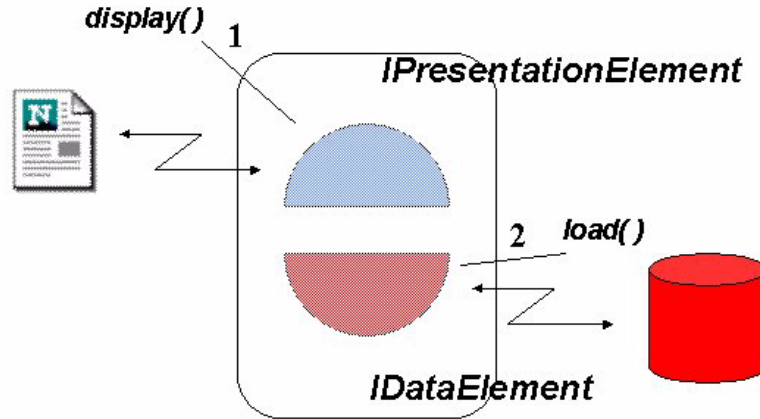
The methods of `IPresentationElement` and `IDataElement` are invoked in a particular order, depending on which of the following actions is performed:

- Displaying a Work Item
- Initiating a Process Instance
- Completing a Work Item
- Accessing a Custom Field from a Script

Displaying a Work Item

When a user displays a work item that includes a custom field, the method invocation is shown in Figure 18.3.

Figure 18.3 Methods invoked when displaying a work item

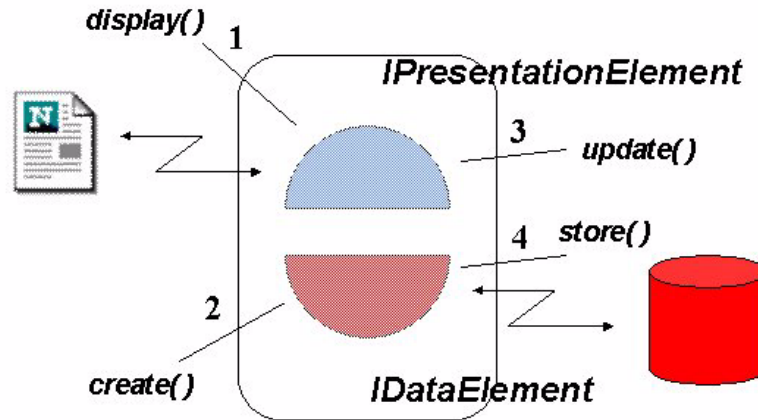


1. The `display()` method (on `IPresentationElement`) displays the custom field in the HTML form.
2. The `load()` method (on `IDataElement`) fetches the value that the field will display.

Initiating a Process Instance

When a user wants to initiate a process instance, the method invocation is shown in Figure 18.4.

Figure 18.4 Methods invoked when initiating a process instance

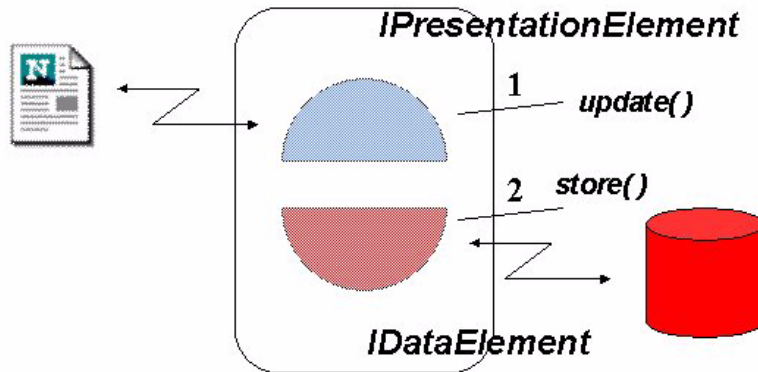


1. The `display()` method (on `IPresentationElement`) displays the custom field at the entry point step. Optionally, the `load()` method (on `IDataElement`) fetches an initial value to display.
2. When the user clicks the submit button, the HTTP POST or HTTP GET request is submitted to PAE. This results in a call to `create()`. Note that all fields have their `create()` method invoked, regardless of whether the field exists on the entry point form.
3. Assuming the field is set for EDIT mode, the field's `update()` method is called.
4. When the process instance is ready to store itself, the field's `store()` method is called. The `store()` method is called only if the field's data was modified by a previous use of `setData()`.

Completing a Work Item

Completing a work item is similar to the entry point form submission, described in the previous case. However, because the process instance already exists, the `create()` method isn't called. The method invocation is shown in Figure 18.5.

Figure 18.5 Methods invoked when completing a work item

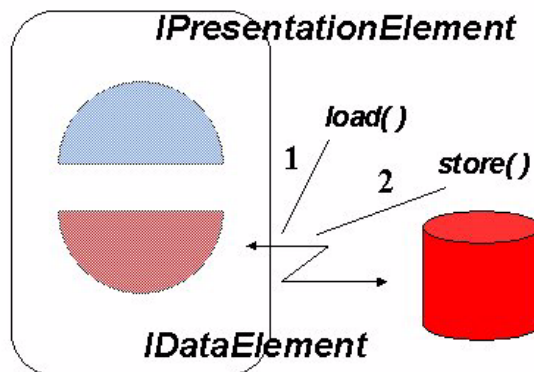


1. Assuming the field is set for EDIT mode, the field's `update()` method is called.
2. When the process instance is ready to store itself, the field's `store()` method is called. The `store()` method is called only if the field's data was modified by a previous use of `setData()`.

Accessing a Custom Field from a Script

A user's JavaScript script might use `getData()` to access the data objects of a custom field. The method invocation for this situation is shown in Figure 18.6.

Figure 18.6 Methods invoked when accessing a custom field from a script



1. The `load()` method is called to fetch the data.
2. The `load()` method typically uses `setData()`. Whenever a `setData()` is performed, the `store()` method is called when the process instance is ready to store itself. As a result, the `store()` method may be called even though the field's data has not changed.

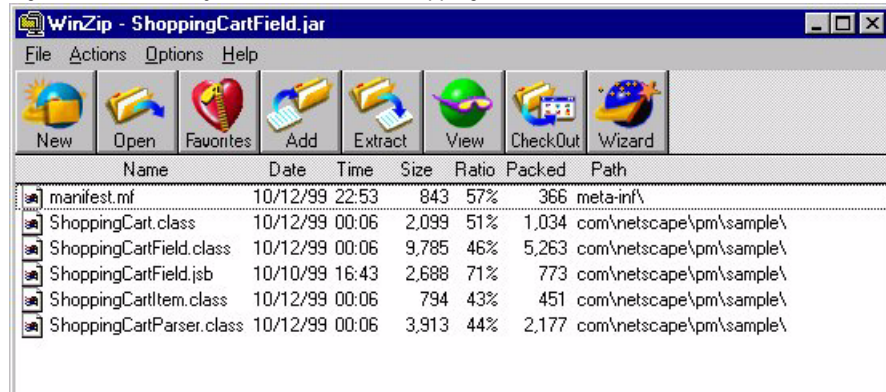
Packaging a Custom Field

After you compile your custom field Java classes and define the JSB file, the next step is to package these files into a zip or jar archive. For example, Figure 18.7 shows an archive that contains the following files:

- `ShoppingCartField.jsb` is the JSB file for this custom field.
- `ShoppingCartField.class` is the class file for this custom field.
- `ShoppingCart.class` and `ShoppingCartItem.class` are class files representing data objects.
- `ShoppingCartParser.class` is an auxiliary class that is used to output data in XML format.

Note When you use the `jar` command to create an archive, a file named `manifest.mf` is automatically created by default. This file contains information about the other files within the archive. The `manifest.mf` file has no effect on the custom field.

Figure 18.7 Directory structure for the ShoppingCartField custom field



Note that the archive file, JSB file, and custom field class must all have the same root name. In the example shown in Figure 18.7, this name is `ShoppingCartField`.

As you create the archive, check that the directory structure reflects the package structure of the class. For example, the class files are in the package `com.netscape.pm.sample`. Therefore, the class files must be in the directory `com/netscape/pm/sample`, as shown in Figure 18.7. The JSB file must be at the same level as the class files.

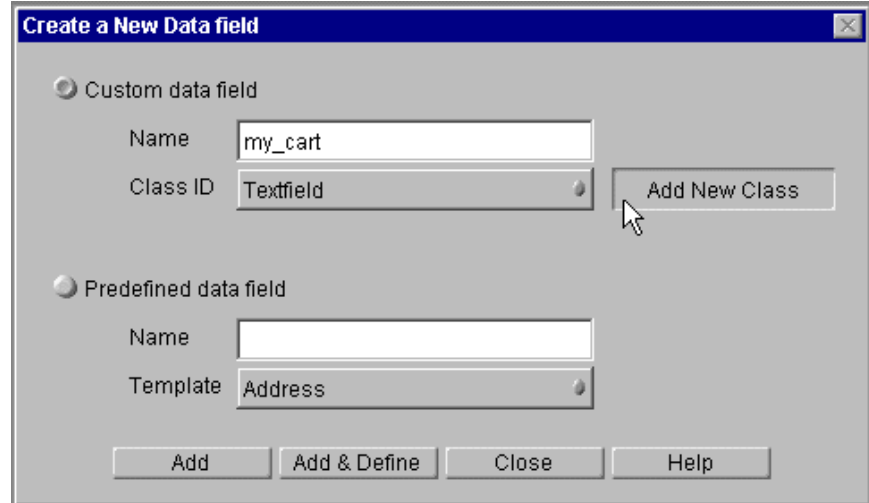
Adding a Custom Field to an Application

After you package a custom field as an archive file, you can add the field in Process Builder, as described in “Creating a Data Field” on page 132.

The specific steps for adding a custom field are as follows:

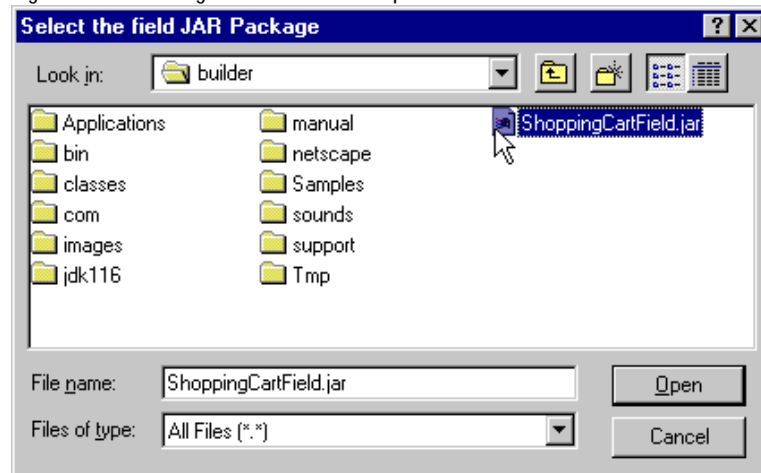
1. From the Insert menu, choose Data Field.
2. In the “Create a New Data Field” dialog box, enter a name for the new field and then click Add New Class. An example is shown in Figure 18.8:

Figure 18.8 Creating a data field from a new class



3. In the “Select the field JAR Package” dialog box, select the archive that represents your custom field class, then click Open. An example is shown in Figure 18.9:

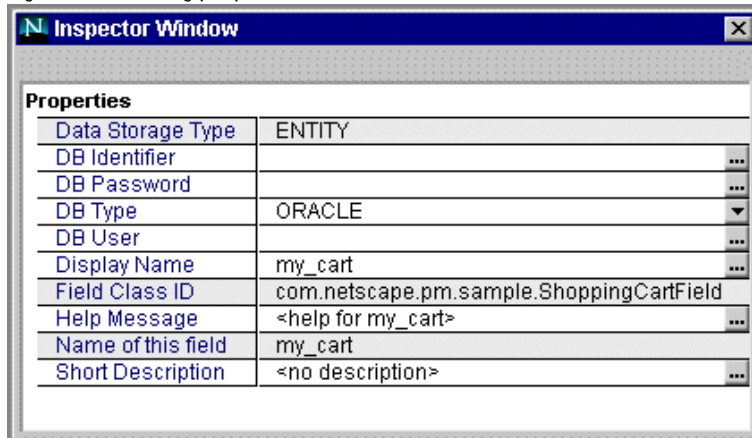
Figure 18.9 Selecting the archive that represents a custom field class



4. In the “Create a New Data Field” dialog box, add the field to the Data Dictionary in either of two ways:

- Click Add to add the field without setting its properties first. The Create a New Data Field dialog box remains open, and you can add more items.
- Click Add & Define to add the field and set its properties immediately. The Inspector window appears for the data field you added, as shown in Figure 18.10

Figure 18.10 Setting properties for the new custom field



5. Set the properties and close the window when you are done.

The new data field, with the properties you defined, now appears in the Data Dictionary folder in the application tree view. You can now use the data field as you would use a typical data field in Process Builder.

Method Reference

This section summarizes the methods available in the following objects:

- IPresentationElement Interface
- IDataElement Interface
- BasicCustomField Class
- IPMElement Interface

IPresentationElement Interface

The IPresentationElement interface has two methods, `display()` and `update()`. Your custom field class must implement these methods, which are summarized in the following sections.

display()

Summary Displays the field in the HTML page.

Syntax 1 This version displays the field after a process instance has been created.

```
public void display(
    IProcessInstance pi,
    IHTMLPage html,
    int displayMode,
    String displayFormat ) throws Exception
```

Syntax 2 This version displays the field when the user is viewing the entry point form.

```
public void display(
    IHTMLPage html,
    int displayMode,
    String displayFormat ) throws Exception
```

Arguments **pi** Object representing the process instance.

html Object representing the HTML page to be returned to the user.

displayMode Mode that the field should be displaying itself in. Possible values are `MODE_EDIT`, `MODE_VIEW` and `MODE_HIDDEN`.

displayFormat Additional formatting information available to the field. This value is specified from the “Display Format” property of the Inspector window of the field when it is placed in the form. This value is specific to a process designer. One possible use is to distinguish between a secure viewing mode and a non-secure viewing mode, such as for credit card information. In such a case, the display format could contain either the value “secure” or “not secure.”

Return Value None.

Description The version of `display()` shown in Syntax 1 will be called after the process instance has been created. In other words, it is called everywhere but the entry point node. The process instance will contain the data that is associated with

your custom field; your implementation of `display()` will need to fetch the data object via the `getData()` method of the process instance class before displaying it.

The `displayMode` and `displayFormat` arguments are defined by the process designer through the Inspector window.

If a call to this method fails, you can throw a `java.lang.Exception` at any time to signal an error. The error message will be displayed to the user.

update()

Summary Translates the HTTP POST or HTTP GET string parameters for this field into the usual data object associated with the field.

Syntax

```
public void update(
    IProcessInstance pi,
    IPMRequest rq ) throws Exception
```

Arguments **pi** Object representing the process instance.

html Object representing the HTTP request.

Return Value None.

Description The `update()` method is called after the user has submitted a request to the PAE server. Since all requests take the form of an HTTP GET or HTTP POST, this method translates the form parameters of the request into the usual data object associated with your custom field. For example, suppose the form includes values for an item ID and an item quantity. The `update()` method would convert the item quantity to a numeric value, and the method would create an `Item` object out of the item ID. The `Item` object could then be bound to the process instance via `setData()`.

If a call to this method fails, you can throw a `java.lang.Exception` at any time to signal an error. The error message will be displayed to the user.

IDataElement Interface

The `IDataElement` interface has four methods: `create()`, `store()`, `load()`, and `archive()`. Your custom field class must implement these methods, which are summarized in the following sections.

create()

Summary Initializes a newly created process instance with a default value for the custom field.

Syntax

```
public void create(
    IProcessInstance pi) throws Exception
```

Arguments **pi** Object representing the process instance.

Return Value None.

Description Most of the time, the `create()` method creates a default value and stores it in the process instance through a call to `setData()`. However, not all custom fields require these actions. This decision is up to the process designer.

If a default value does not need to be set, it is recommended that you do not implement the `create()` method. Leave it blank instead. The `store()` method for custom fields is called only when `setData()` has been performed on the field.

The `create()` method for all fields, whether predefined or custom fields, is called when the user initiates a process instance from the entry point.

If a call to this method fails, you can throw a `java.lang.Exception` at any time to signal an error. The error message will be displayed to the user, and the process instance will not be created.

store()

Summary Stores the data associated with the custom field to a persistent resource.

Syntax

```
public void store(
    IProcessInstance pi) throws Exception
```

Arguments **pi** Object representing the process instance.

Return Value None.

Description It's up to the designer of the custom field to decide which external persistent datastore will store the custom field data. Note, however, that data from a custom field cannot be stored in the application-specific table, where built-in data fields are stored.

The custom field is responsible for storing the data, whereas PAE is responsible for storing the custom field's primary key. This key is stored in the application-specific database table.

The `store()` method is called only if the field's value has been modified, through a call to `setData()`. Note that the `load()` method typically calls `setData()`. As a result, the `store()` method is called whenever `load()` is called.

Currently, PAE does not support global transactions. If the custom field stores its data in an external datasource that is both XA-compliant and managed by a resource manager, the custom field could participate in a global transaction. However, transactions initiated by PAE are not made through an XA resource manager, so they cannot be a part of the larger transaction.

If a call to this method fails, you can throw a `java.lang.Exception` at any time to signal an error. The current work item is converted to an exception work item, and all data field values are reset to their values prior to the request.

load()

Summary Loads, from a persistent resource, the data associated with the custom field.

Syntax

```
public void load(
    IProcessInstance pi) throws Exception
```

Arguments **pi** Object representing the process instance.

Return Value None.

Description The `load()` method is invoked whenever the data value associated with the custom field is accessed through `getData()` off the process instance. Note that built-in fields are loaded whenever the process instance is loaded, but custom fields are loaded only when their data value is explicitly asked for. This behavior is called lazy loading.

Warning: Within the `load()` method, do not call `getData()` on the custom field. The `load()` method is already invoked as a result of a call to `getData()`. As a result, a call to `getData()` within the `load()` method causes an infinite loop.

If a user script accesses or modifies the data associated with a custom field, the script must implicitly know the object's data type. For example, a script would need to know the API for objects such as `ItemSet` and `Item` in a shopping cart custom field.

If a call to `load()` fails, you can throw a `java.lang.Exception` at any time to signal an error. If the current action is to display a form, an error message will be displayed to the user. If the user has completed a work item, an exception work item will be created.

archive()

Summary Writes the data associated with the custom field to an output stream.

Syntax

```
public void update(
    IProcessInstance pi,
    OutputStream os) throws Exception
```

Arguments **pi** Object representing the process instance.

os The output stream to write the data to.

Return Value None.

Description When an archive operation is initiated from the administration pages, the data elements associated with the process instance write their data values to an output stream. Built-in data elements archive themselves, simply by writing their values out as bytes. By contrast, you can determine how custom fields write their data to an output stream. For example, you can stream bytes or encapsulate the values in XML.

If a call to this method fails, you can throw a `java.lang.Exception` at any time to signal an error. The error message will be displayed to the administrator.

BasicCustomField Class

By implementing the methods on `IPresentationElement` and `IDataElement`, you define how your custom field will manage its data objects. However, there are additional methods to implement so that PAE treats your custom field just like any other data field. PAE provides these method implementations for you, through the `BasicCustomField` class.

This class also provides empty stubs for the `IPresentationElement` and `IDataElement` methods that you must fill in.

Therefore, to write your custom field class, use `BasicCustomField` as your base class. For example:

```
import com.netscape.pm.model.BasicCustomField;

public class myCustomField
    extends BasicCustomField
{
    ...
}
```

In addition, `BasicCustomField` provides the `loadDataElementProperties()` method, described next.

loadDataElementProperties()

Summary Loads the design-time properties for the field specified in Process Builder's Inspector window.

Syntax `protected void loadDataElementProperties(Hashtable entry) throws Exception`

Arguments **entry** The hashtable containing field configuration properties.

Return Value None.

Description This method is called after the custom field has been created (while the application is being initialized). The hashtable entry parameter contains the field's configuration information, as it is stored in the LDAP repository. This information includes the properties you specified in the custom field's JSB file.

If a call to this method fails, you can throw a `java.lang.Exception` at any time to signal an error. The error message will be displayed to the user, and the application will stop being initialized.

Example Suppose your JSB file contains the following entry:

```
<JSB_PROPERTY
  NAME="dbidentifier"
  TYPE="string"
  DISPLAYNAME="External DB Identifier"
  SHORTDESCRIPTION="Local alias used to connect to external DB server"
  ISEXPERT>
```

Given the previous JSB code, the following Java code implements the `loadDataElementProperties()` method. The method will first read the property and then, based on the value, set the instance variable `mDBIdentifier`.

```
protected void loadDataElementProperties( Hashtable entry )
    throws Exception
{
    String dbIdentifier = (String) entry.get( "dbidentifier" );
    if( dbIdentifier == null )
        throw new Exception( "DB Identifier not specified" );
    else
        mDBIdentifier = dbIdentifier;
}
```

IPMElement Interface

The `BasicCustomField` class implements the `IPMElement` interface, which has two methods: `getName()` and `getPrettyName()`. These methods make it easier to implement your custom field.

getName()

Summary Returns the name of the current element.

Syntax `public String getName()`

Arguments None.

Return Value A `String` object representing the name of the current element.

Description The returned name is used to access the field's primary key and data value.

Example The following code uses `getName()` inside the `create()` method:

```
public void create( IProcessInstance pi )
    throws Exception
{
    // Assign a default value for this field.
    Just an empty shopping cart...
    //
    pi.setData( getName(), new ShoppingCart() );
}
```

getPrettyName()

Summary Returns the “pretty name” of the current element.

Syntax `public String getPrettyName()`

Arguments None.

Return Value A String object representing the pretty name of the current element.

Description In previous releases of PAE, every element had a name as well as a “pretty name,” the display name of the element. In the current release, an element’s pretty name and its name are equivalent.



JavaScript API Reference

This appendix is an API reference of the objects and methods you can use when writing JavaScript scripts for PAE. The PAE Object Model includes the following objects:

- `ProcessInstance` contains methods for accessing information about the current process.
- `WorkItem` contains methods for accessing information about the current work item.
- `ContentStore` contains methods for accessing the content store.
- `CorporateDirectory` contains methods for accessing data about users in the Directory Server.
- `User` exposes the user attributes that make up an LDAP user entry as publically accessible properties.

The following global functions are also available for use in scripts:

- Logging and Error Handling Global Functions
- Assignment, Completion, and Email Scripts
- Miscellaneous Global Functions

All functions and methods are listed here:

- Alphabetical Summary of JavaScript Objects

ProcessInstance

A script can use the global function `getProcessInstance()` to get the processInstance for process in which the script is running.

Example:

```
var pi = getProcessInstance ();
```

The following methods can be called on the processInstance object:

- `getConclusion`
- `getCreationDate`
- `getCreatorDN`
- `getCreatorUser`
- `getData`
- `getEntityKey`
- `getEntryNodeName`
- `getExitNodeName`
- `getInstanceId`
- `getPriority`
- `getRoleDN`
- `getRoleUser`
- `getTitle`
- `setData`
- `setEntityKey`
- `setPriority`
- `setRoleByDN`
- `setRoleById`
- `setTitle`

getConclusion()

Returns the name of the exit point node where the process instance is completed. Data is returned as a JavaScript string if the process instance is complete. If the process instance is still active, `null` is returned.

Example:

```
var pi = getProcessInstance();
var exitNodeName = pi.getConclusion();
if( exitNodeName != null )
    java.lang.System.out.println( "Process instance completed at: " +
    exitNodeName );
```

It is recommended that you use `getExitNodeName` rather than `getConclusion` as `getConclusion` may be deprecated in future releases.

getCreationDate()

Returns a JavaScript date that represents the date when the process instance was first created.

Example:

```
var pi = getProcessInstance();
java.lang.System.out.println( "Process instance created on: " +
pi.getCreationDate() );
```

getCreatorDN()

Returns the distinguished name (DN) of the user who first created the process instance.

Example:

```
"uid=topper, o=airius.com"
```

getCreatorUser()

Returns an object that contains the attributes of the creator user in the corporate directory.

Example:

```
var pi = getProcessInstance ();
var creator = pi.getCreatorUser ();
java.lang.System.out.println( "Process instance creator id: " +
creator.getUserID() );
```

getData (fieldName)

Returns the value of the data field or undefined if the field is not defined. The returned value is in JavaScript format.

Parameter	Data Type	Description
fieldName	String	Name of the field whose value is returned.

Example:

```
var pi = getProcessInstance ();
var cust_name = pi.getData( "customer_name" );
java.lang.System.out.println( "The value of Customer Name: " + cust_name
);
```

getEntityKey(fieldName)

Returns the entity key associated with the custom field whose name is passed as an argument. This key is stored in the process instance and is set using the `setEntityKey` method. The entity key is the primary key for the field, uniquely identifying it from other instances of the same custom field.

Parameter	Data Type	Description
<code>fieldName</code>	String	Name of the custom field for which you want to fetch the entity key.

Example:

```
var pi = getProcessInstance() ;
pi.getEntityKey("mySignature") ;
```

getEntryNodeName()

Returns a JavaScript string containing the name of the entry point node where the current process instance was initiated. All process instances must start from an entry node, ensuring that this method never returns null.

Example:

```
var pi = getProcessInstance();
java.lang.System.out.println( "Process instance entry node is " +
pi.getEntryNodeName() );
```

getExitNodeName()

Returns a JavaScript string containing the name of the exit point node where the process instance has completed. If the process instance has not yet completed, returns null.

Example:

```
var pi = getProcessInstance();
var exitNodeName = pi.getExitNodeName();
if( exitNodeName != null )
    java.lang.System.out.println( "Process instance completed at: " +
exitNodeName );
```

getInstanceId ()

Returns the ID of the process instance, such as "2345". The id is generated by the engine to be unique across all process instances of all applications defined in the same cluster.

Example:

```
var pi = getProcessInstance();
var pid = pi.getInstanceId();
java.lang.System.out.println( "The Process Instance ID: " + pid );
```

getPriority()

Returns a JavaScript number representing the priority value of the current process instance. The priority value ranges from 1 (highest) to 5 (lowest). The priority value is derived from the value of a data element. If the data element's type is INT/FLOAT, the priority value is fetched from the data element's value. Otherwise, the default value 3 is returned.

Example:

```
var pi = getProcessInstance();
var priority = pi.getPriority();
if( priority == 1 )
    java.lang.System.out.println( "The process instance has high
priority" );
else if( priority == 5 )
    java.lang.System.out.println( "The process instance has low
priority" );
else
    java.lang.System.out.println( "The current priority is " + priority
);
```

getRoleDN(roleName)

Returns a string containing the distinguished name (DN) of the user that has been assigned to the role *roleName*. You can fetch the DN for the current user using this method.

Parameter	Data Type	Description
roleName	String	Name of the role whose full DN is returned.

Example:

```

var pi = getProcessInstance ();
var cd = getCorporateDirectory();
var userDN = pi.getRoleUser( "role1" 0;
if( userDN != null)
{
    var u = cd.getUserByDN( userDN );
    java.lang.System.out.println( "Role user id: " + u.getUserId() );
}

```

getRoleUser(roleName)

Returns a JavaScript User object that contains the attributes of the user that has been assigned to the role `roleName`. If a user is not associated with the role `roleName` yet, null is returned.

Parameter	Data Type	Description
<code>roleName</code>	String	Name of the role whose full DN is returned.

Example:

```

var pi = getProcessInstance ();
var u = pi.getRoleUser( "role1" );
if(u != null )
    java.lang.System.out.println( "role user id: " + u.getUserId() );

```

getTitle()

Returns the current title string associated with the process instance. Calling this method is similar to calling `getData()` on the title data field and casting the value to a string. If the title data field has a type other than TEXT/LONG TEXT, returns a textual representation of the data element's value.

Example:

```

var pi = getProcessInstance();
java.lang.System.out.println( "The current title is: " + pi.getTitle() );

```

setEntityKey(fieldName, value)

Sets the entity key of a custom field. The entity key is the primary key for the field, uniquely identifying it from other instances of the same custom field. The entity key is always available from the process instance, and is stored with the rest of the application data.

Parameter	Data Type	Description
<code>fieldName</code>	String	Name of the custom field you want to modify.
<code>value</code>	Object	Value of the custom field's entity key.

Example:

```
var pi = getProcessInstance() ;
pi.setData ( "mySignature" , pi.getData("signature") ) ;
pi.setEntityKey("mySignature" , "bob") ;
```

setData (fieldName, fieldValue)

Associates the object *fieldValue* with the data element *fieldName*. The object passed by *fieldValue* can be any valid JavaScript object (such as String, Date, Array). The object can be a Java object if the data element is a custom field.

Parameter	Data Type	Description
<code>fieldName</code>	String	Name of the data element whose value is modified.
<code>fieldValue</code>	Object	Value associated with the data element <code>fieldName</code> .

Since each data element is configured with a particular data type which is set at design time in Process Builder, be aware of the following data conversion guidelines:

Data element type	fieldValue type	fieldValue converted to...
TEXT/LONGTEXT	any type	String
INT	String	int

Data element type	fieldValue type	fieldValue converted to...
INT	Integer	int
FLOAT	String	float
FLOAT	Integer	float
DATE/DATETIME	String	Date
DATE/DATETIME	Date	Date

Example:

```
var pi = getProcessInstance ();
pi.setData( "customer_name", "Joe" );
//Set value for custom field
//
var custAcct = new Packages.com.acme.Account();
pi.setData( "customer_acct", custAcct );
```

setPriority(value)

Changes the current priority value for the process instance to the integer value *value*. A valid priority value can range from 1 (highest) to 5 (lowest). If a value smaller than 1 is passed, the priority is set to 1. If a value larger than 5 is passed, the priority is set to 5. If a non-integral value is passed through *value*, an attempt is made to convert that value to an integer. If the value cannot be converted to an integer, the priority is left unchanged.

Parameter	Data Type	Description
value	Integer	Integer value set as the current priority for the process instance.

Example:

```
var pi = getProcessInstance();
pi.setPriority( 3.141592 ); //priority := 3
pi.setPriority( -1 ); //priority := 1
pi.setPriority( "99" ); //priority := 5
```


setRoleByDN(roleName, userDN)

Changes the user associated with the role *roleName* by specifying the new user's Distinguished Name (DN) *userDN*. If *userDN* corresponds to an actual user in the corporate directory, the user associated with *roleName* is changed and `true` is returned. If the *userDN* does not correspond to an actual user entry, `false` is returned and *roleName* is left unchanged.

Parameter	Data Type	Description
<code>roleName</code>	String	Name of the role to alter.
<code>userDN</code>	String	DN of the user to associate with <code>roleName</code> .

Example:

```
var pi = getProcessInstance ();
if( pi.setRoleByDN( "uid=joe, ou=People, o=acme.com" ) )
    java.lang.System.out.println( "role has been changed" );
else
    java.lang.System.out.println( "role has not been changed" );
```

setRoleById(roleName, userId)

Changes the user associated with the role *roleName* by specifying the new user's ID via *userId*. Returns `true` if *userId* corresponds to an actual user in the corporate directory and the user associated with *roleName* is changed. Returns `false` if *userId* does not correspond to an actual user entry and *roleName* is left unchanged.

Parameter	Data Type	Description
<code>roleName</code>	String	Name of the role to alter.
<code>userDN</code>	String	DN of the user to associate with <code>roleName</code> .

Example:

```
var pi = getProcessInstance ();
if( pi.setRoleById( "joe" ) )
    java.lang.System.out.println( "role has been changed" );
else
    java.lang.System.out.println( "role has not been changed" );
```

setTitle(title)

Changes the current title for the process instance to the value contained in *title*.

title can be any valid JavaScript object, but the resulting title is the textual representation of the passed object. Calling `setTitle(title)` does not change the value of the title data element; the binding is uni-directional from the data element to the process instance title.

Parameter	Data Type	Description
<code>title</code>	Object	Value set as the current title for the process instance.

Example:

```
var pi = getProcessInstance();
pi.setTitle( "a new title" ); //title := "a new title"
pi.setTitle( new Date() ); //title := "Tue Oct 19 13:04:50 CDT 1999"
pi.setTitle( 3.141592 ); //title := "3.141592"
```

WorkItem

A script can use the global function `getWorkItem()` to get the work item in which the script is running.

Example:

```
var wi = getWorkItem();
```

The following methods can be called on the `workItem` object:

- `assignTo`
- `expire`
- `extend`
- `getAssigneesDN`
- `getCreationDate`
- `getCurrentActivityCN`
- `getExpirationDate`
- `hasExpired`
- `isActive`
- `isRunning`
- `isStateSuspended`
- `moveTo`
- `resume`
- `setExpirationDate`
- `suspend`

assignTo(assigneeDnArray)

Re-assigns the work item to the users specified in the `assigneeDnArray`, which must be an array of valid distinguished names (DNs). This function automatically sets the work item back to running if it has expired.

Parameter	Data Type	Description
<code>assigneeDnArray</code>	Array	List of assignee DN's to assign to the work item.

The following script re-assigns the work item to the user whose uid is `jocelyn`. The `toUserId()` function returns an array containing the distinguished name of the user whose uid is specified. For more information about `toUserId()` see the section "Miscellaneous Global Functions."

Example:

```
var wi = getWorkItem();
var cd = getCorporateDirectory();
var joe = cd.getUserById( "joe" );
var bob = cd.getUserById( "bob" );
var assignees = new Array( joe.getDN(), bob.getDN() );
wi.assignTo( assignees );
```

expire()

This function adds a flag indicating that the work item has expired. The function does not render the work item inactive.

Example:

```
var wi = getWorkItem();
wi.expire();
if( wi.hasExpired() )
    java.lang.System.out.println( "The workitem has expired" );
else
    ;
```

extend(newDate)

Extends the duration of the work item by pushing out its expiration date. If the work item was expired, this function removes the expired flag.

Parameter	Data Type	Description
newDate	Date	New expiration date for the work item.

Example:

```
var wi = getWorkItem();
var now = new Date();

// Set extension date to 1 hour in the future
var newDate = new Date( now.getTime() + (1 * 60 * 60 * 1000) );
wi.extend( newDate );
```

getAssigneesDN()

For work items that are running, this method returns a JavaScript array of the distinguished names of the assignees. Since automated activities are performed by Process Engine and do not require an assignee, use this method only from a user activity.

Example:

```
var wi = getWorkItem();
var cd = getCorporateDirectory();

//Set the assignees for the workitem

var joe = cd.getUserById( "joe" );
var bob = cd.getUserByid( "bob" );
var assignees = new Array( joe.getDN(), bob.getDN() );
wi.assignTo( assignees );

var nAssignees = wi.getAssigneesDN();
for( var i = 0; i < nAssignees.length; i++)
    java.lang.System.out.println( "Assignee " + i + " is " + nAssignees[
i ] );
```

getCreationDate()

Returns a JavaScript date that represents the creation date of the work item.

Example:

```
var wi = getWorkItem();
java.lang.System.out.println( "Work item created on: " +
wi.getCreationDate() );
```

getCurrentActivityCN()

Returns a JavaScript string containing the name of the activity where the work item is positioned. When called from an exception activity, returns the name of that exception activity rather than the name of the activity where the exception condition occurred.

Example:

```
var wi = getWorkItem();
java.lang.System.out.println( "Work item positioned at node " +
wi.getCurrentActivityCN() );
```

getExpirationDate()

Returns a JavaScript `Date` object that represents the expiration date of the work item.

Example:

```
var wi = getWorkItem();
java.lang.System.out.println( "Work item set to expire at " +
wi.getExpirationDate() );
```

hasExpired()

Returns `true` if the work item has expired or `false` if it has not expired. If the work item has already expired, but has been extended via the `extend()` method, `hasExpired()` returns `false` if called again.

Example:

```
var wi = getWorkItem();
if( wi.hasExpired() == false )
    wi.expire(); //expire immediately
```

isStateActive()

Returns `true` if the work item is active or `false` if the work item is inactive. A work item is considered active if there is only one assignee currently assigned to it. If a work item has been assigned to a group, the work item is not active, but running. Once one member of the group elects to work on the item, it is considered active.

Example:

```
var wi = getWorkItem();
if( wi.isStateActive() 0
    java.lang.System.out.println( "Work item is active" );
else
{
    var nAssignees = wi.getAssigneesDN();
    if( nAssignees.length > 1 )
        java.lang.System.out.println( "Work item is running"
);
    else
        ; //work item could be suspended
}
```

isStateRunning()

Returns `true` if the work item is running or `false` if it is not running. A work item is running if more than one user is assigned to it. Once a user starts the work item, the work item changes state from running to active.

Example:

```
var wi = getWorkItem();
if( wi.isStateRunning() )
{
    var nAssignees = wi.getAssigneesDN*();
    if( nAssignees.length > 1 )
        java.lang.System.out.println( "Work item is assigned to a group" );
    else
        ;
}
```

isStateSuspended()

Returns `true` if the work item is suspended or `false` if it is not suspended. You can suspend a work item through Process Administrator or by calling `suspend()`.

Example:

```
var wi = getWorkItem();
if( wi.isStateSuspended() )
    wi.resume(); //resume the work item immediately
```

moveTo(activityName)

Move the current work item to the activity specified in *activityName*. Only activities associated with a virtual transition can use this method. `moveTo()` can only be called from expiration handler scripts and completion scripts.

Note that `moveTo()` doesn't actually involve "moving." Rather, the current work item is destroyed and a new work item is created at the new activity.

Parameter	Data Type	Description
<code>activityName</code>	String	Activity to which to move the work item.

Example:

```
var wi = getWorkItem(); //inside the expiration handler script
var activityName = "expired item";
if( wi.moveTo( activityName ) )
    java.lang.System.out.println( "Successfully moved to activity " +
activityName );
else
    java.lang.System.out.println( "Failed to move to activity " +
activityName );
```

resume()

This function reactivates a suspended work item. If the work item is assigned to more than one user, the work item's state is set to running. If there is only one assignee, the work item's state is set to active.

Example:

```

var wi = getWorkItem();
if( wi.isStateRunning() )
{
    wi.resume(); //resume work item immediately
    if( wi.isStateRunning() )
        java.lang.System.out.println( "Work item is assigned to a group: );
    else
        java.lang.System.out.println( "work item is assigned to one user"
);
}

```

setExpirationDate(expDate)

Sets the expiration date of the work item. If the work item has expired, use the `extend()` function instead of `setExpirationDate()` to extend the expiration date, since `extend()` removes the expired flag and `setExpirationDate()` does not.

Parameter	Data Type	Description
expDate	Date	Expiration date for the work item.

Example:

```

var wi = getWorkItem();
var now = new Date();

//Set expiration date to 1 hour in the future

var expDate = new Date( now.getTime() + (1 * 60 * 60 * 1000)
);
wi.setExpirationDate( expDate );

```

suspend()

This function makes the work item inactive. Users cannot work with the work item until it has been resumed through Process Administrator or via a call to `resume()`.

Example:

```

var wi = getWorkItem();

```



```

wi.suspend();
if( wi.isStateSuspended() )
    wi.resume(); //resume the work item immediately

```

ContentStore

Scripts can use the global function `getContentStore()` to get a `contentStore` object connected to the content store of the application. There are three different ways to call the function:

- `getContentStore ()`;
Without parameters, this function returns the content store defined by the application. The returned object is configured with the URL, username, and password that are specified in Process Builder.
- `getContentStore (httpURL, username, password)`;
When called with the above three parameters, `getContentStore()` creates a new content store with the specified parameters passed in. The returned object is a content store rooted at the specified `httpURL`. The content store uses the username and password to authenticate against.
- `getContentStore (httpURL, username, password, timeout)`;
This form is identical to the previous form except for an additional timeout, specified in seconds, on HTTP connections.

The following methods can be called on the `contentStore` object:

- | | | |
|----------------|---------------|----------|
| • copy | • getSize | • move |
| • download | • getStatus | • remove |
| • exists | • getVersion | • rmdir |
| • getBaseURL | • initialize | • store |
| • getContent | • isException | • upload |
| • getException | • list | |
| • getRootURL | • mkdir | |

copy(srcURL, dstURL)

Copies the content at the *srcURL* to the *dstURL*. Both URLs have to be on the same content store. If *dstURL* already exists, it is overwritten.

Parameter	Data Type	Description
srcURL	String	An HTTP URL or URI of the content in the content store object.
dstURL	String	An HTTP URL or URI of the content in the content store object.

Example:

```
var cs = getContentStore();

var result = cs.copy( cs.getBaseURL ( "shopping/documents" ) +
"myFile.html",
cs.getBaseURL ( "shopping/old" ) + "myFile.html" );
var httpcode = cs.getStatus ( result );
if (httpcode == 200)
{
}
```

download(url, file)

Downloads the content at the url into the specified file. If a username and password is specified for this content store, they are used in the request to retrieve the content.

Parameter	Data Type	Description
url	String	An HTTP URL or URI of the content in the content store object.
file		File where the content is stored; if exists, it will be overwritten.

Example:

```
var cs = getContentStore();
var result = cs.download ( cs.getBaseURL() + "myFile.html" , "C:/
```

```

myFile.html" );
var httpcode = cs.getStatus( result );
if (httpcode == 200)
{
// it went OK ...
}

```

exists(url)

Checks if the URL exists on the content store.

Parameter	Data Type	Description
url	String	An HTTP URL or URI of the content in the content store object.

Example:

```

var cs = getContentStore();

var result = cs.exists( cs.getBaseURL ( "shopping/documents" ) +
"myFile.html" );
var httpcode = cs.getStatus ( result );
if (httpcode == 200)
{
// it exists OK ...
}

```

getBaseURL()

Returns the root URL of this content store.

See `getRootURL ()` for examples.

getBaseURL(path)

Returns a base container url which includes the path as postfix to the content store root url.

Parameter	Data Type	Description
path	String	A relative path segment.

Example:

```
var cs = getContentStore();
java.lang.System.out.println("ContentStore ROOT URL " +
cs.getBaseURL("shopping/documents"));
```

getBaseURL(path, instanceid)

Returns a base container URL which includes the path and the `instanceid` as postfix to the content store root URL.

Parameter	Data Type	Description
<code>path</code>	String	A relative path segment.
<code>instanceid</code>	Integer	An integer instance id of the process.

Example:

```
var pi = getProcessInstance();
var cs = getContentStore();
java.lang.System.out.println("ContentStore ROOT URL " +
cs.getBaseURL("shopping/documents", pi.getInstanceId() ));
```

getContent (url)

Returns a String of the content at the specified url, where `url` describes either an HTTP URL or URI of the content in the content store object. A URI is a path segment only, specified as an absolute or relative pathname.

Parameter	Data Type	Description
<code>url</code>	String	An HTTP URL or URI of the content in the content store object.

On an HTTP content store, `getContent()` is implemented as a GET request against the server. If a username and password have been specified for this content store, they will be used in the request to retrieve the content.

If the content cannot be retrieved or doesn't exist, `getContent()` returns null. Call the `exists()` method to check whether the URL exists in the content store. See also `store()` and `upload()`.

Example:

```

var cs = getContentStore();
// Relative URI
var content = cs.getContent ( "myFile.html" );
// Absolute URI
var content = cs.getContent ( "/CS/CheckOut/basket.xml" );
// If URL, then the host and port portion of the URL must be
// that of the content store URL. This is typically returned
// by the getBaseURL() method.
var content = cs.getContent ( cs.getBaseURL() +
"CheckOut/basket.xml" )

```

getException(result_string)

Gets the exception in string format that has occurred in the operation `move()`, `copy()`, `upload()`, `download()`, `store()`, `exists()`, `initialize()`, `remove()`, `mkdir()`, `rmdir()`.

Parameter	Data Type	Description
<code>result_string</code>	String	Result string from one of the methods listed above.

Example:

```

var cs = getContentStore();

var result = cs.remove( cs.getBaseURL ( "shopping/documents" ) +
"myFile.html" );
if ( cs.isException ( result ) )
{
// it something went really wrong
java.lang.System.out.println("Exception: " + cs.getException ( result
));
}
else if ( cs.getStatus ( result ) )
{
} else if ( cs.getStatus ( result ) )
{
}
}

```

getRootURL ()

Returns a string of the root URL of the content store, including the ending /. This is a string version of the URL for the content store as specified in Process Builder.

Example:

```
var cs = getContentStore();
java.lang.System.out.println("ContentStore ROOT URL " +
cs.getRootURL());
```

getSize(url)

Returns the size in bytes of the content of the specified URL at the content store. On any error, -1 is returned.

Note that in certain rare circumstances, the length of the content of such a URL cannot be established.

Parameter	Data Type	Description
url	String	An HTTP URL or URI of the content in the content store object.

Example:

```
var cs = getContentStore();

var result = cs.getSize( cs.getBaseURL ( "shopping/documents" ) +
"myFile.html" );
if (result < 0)
{
}
```

getStatus(result)

Returns the status of the last successful Content Store operation by one of the methods: move(), copy(), upload(), download(), store(), exists(), initialize(), remove(), mkdir(), rmdir().

Parameter	Data Type	Description
result	String	Result string from one of the methods listed above.

Example:

```
var cs = getContentStore();
var result = cs.remove( cs.getBaseUrl ( "shopping/documents" ) +
"myfile.html" );
if ( cs.getStatus(result) != 200)
{
}
```

getVersion()

Returns the version string of the the ContentStore implementation. Currently, this value is 0.1999100221-001.

Example:

```
var cs = getContentStore();
java.lang.System.out.println("ContentStore Version " +
cs.getVersion());
```

initialize(url)

Initializes the URL with basic content so the URL exists on the Content Store. This method returns either the status code from the content store for the operation or an exception. Note that no exception is actually thrown. To see if an exception is returned, call `isException()` on the return value from this method and call `getStatus()` to get the numeric HTTP protocol status of this content store request.

Parameter	Data Type	Description
url	String	An HTTP URL or URI of the content in the content store object.

Example:

```

var pi = getProcessInstance();
var cs = getContentStore();

var result = cs.initialize ( cs.getBaseUrl ( "shopping/documents" ) +
"myFile.html" );
var httpcode = cs.getStatus ( result );
if (httpcode == 201 || httpcode == 204)
{
}
else
{
}
}

```

isException(exception)

Checks if the result string from `move()`, `copy()`, `upload()`, `download()`, `store()`, `exists()`, `initialize()`, `remove()`, `mkdir()`, `rmdir()` is an exception.

Parameter	Data Type	Description
exception	String	The result string from methods of Content Store that can return an exception.

Example:

```

var cs = getContentStore();

var result = cs.remove( cs.getBaseUrl ( "shopping/documents" ) +
"myFile.html" );
if ( cs.isException ( result ) )
{
}
else if ( cs.getStatus ( result ) )
{
}
}

```

list(url)

Gets the container content (folder) listing of this content store container.

Parameter	Data Type	Description
url	String	An HTTP URL or URI of the content in the content store object.

mkdir (URLString)

Creates the appropriate folder structure on the web server being used as the content store. The *URLString* must be a full path name to the new directory. If the specified directory already exists, nothing happens.

Parameter	Data Type	Description
URLString	String	An HTTP URL or URI of the content in the content store object.

Example:

```
var cs = getContentStore();

var result = cs.mkdir( cs.getBaseURL ( "shopping/documents" ) );
var httpcode = cs.getStatus ( result );
if (httpcode == 200)
{
}
```

move (String URLString1, String URLString2)

moves the file from the *URLString1* to *URLString2*, overwriting any file that already exists at the destination file name. If the file to be moved *URLString1*, does not exist, this function gives an error.

Both file names must be full pathnames. The following code moves the file `tempdoc.html` from the `temp` directory to `doc1.html` in the `docs` directory on the web server `pm.company.com`.

Example:

```
var myStore = getContentStore ();
var pi = getProcessInstance ();
var pid = pi.getInstanceId();
```

```
myStore.move( "http://pm.company.com/temp/tempdoc.html",
             "http://pm.company.com/docs/doc1.html" );
```

Note that the `move()` function does not create directories -- use `mkdir()` to create a directory before moving a file to it.

remove(url)

Deletes the content at the specified URL.

Parameter	Data Type	Description
url	String	An HTTP URL or URI of the content in the content store object.

Example:

```
var cs = getContentStore();

var result = cs.remove( cs.getBaseURL ( "shopping/documents" ) +
                       "myfile.html" );
var httpcode = cs.getStatus ( result );
if (httpcode == 200)
{
}
```

rmdir(url)

Removes the container (folder) at the specified URL. The folder must be empty to be removed or an error is returned.

Parameter	Data Type	Description
url	String	An HTTP URL or URI of the content in the content store object.

Example:

```
var cs = getContentStore();

var result = cs.rmdir( cs.getBaseURL ( "shopping/documents" ) );
var httpcode = cs.getStatus ( result );
```

```

if (httpcode == 200)
{
}

```

store (content, url)

Stores the content specified by *content* in the given url. If a username and password has been specified for this content store, they are used in the request to store the content.

Parameter	Data Type	Description
content	String	Content to store in the content store as this URL.
url	String	An HTTP URL or URI of the content in the content store object.

Example:

```

var cs = getContentStore();

var content = "<body><b>Hello</b> world ... </body>";
// Relative URI
var result = cs.store ( content, "myFile.html" );
// Absolute URI
var result = cs.store ( content, "/CS/myFile.html" );
// Absolute URL (for historical reasons)
var result = cs.store ( content, cs.getBaseURL () + "myFile.html" );

```

upload(file, url)

Uploads the file content to the URL specified. If a username and password has been specified for this content store, they are used in the request to upload the file.

Parameter	Data Type	Description
file		File to upload.
url	String	An HTTP URL or URI of the content in the content store object.

Example:

```

var cs = getContentStore();
var result = cs.upload ( "C:/myFile.html", cs.getBaseURL() +
"myFile.html" );
var httpcode = cs.getStatus( result );
if (httpcode == 201 || httpcode == 204)
{
}

```

CorporateDirectory

Scripts can use the global function `getCorporateDirectory()` to get a `corporateDirectory` object that is connected to the corporate directory in the Directory Server for the cluster.

Example:

```
var corp = getCorporateDirectory ();
```

Scripts can then use the `corporateDirectory` object to access and modify information about users in the corporate directory.

The following methods can be called on the `corporateDirectory` object:

- `addUser`
- `deleteUserByDN`
- `deleteUserByCN`
- `deleteUserById`
- `getUserByCN`
- `getUserByDN`
- `getUserById`
- `modifyUserByCN`
- `modifyUserByDN`
- `modifyUserById`

addUser (userDN, attributes, objectClasses)

Adds a user to the directory. The user's Distinguished Name (DN) is specified by `userDN`. The attributes for the user are specified as a JavaScript associative array indexed by attribute name. The `objectClasses` parameter specifies any additional object classes as a JavaScript array. If the addition of the user is successful, returns `true`, otherwise returns `false`.

Parameter	Data Type	Description
<code>userDN</code>	String	DN for the new user entry.

attributes	Object	Table of key/value pairs specifying user attributes.
objectClasses	Array	List of object classes with which the user entry is compliant.

Example:

```

var attrs = new Object();
attrs[ "uid" ] = "joe";
attrs[ "cn" ] = "Joe Cool";
attrs[ "sn" ] = "Cool";
attrs[ "mail" ] = "joe@acme.com";
attrs[ "favoriteColor" ] = "green";
// Specify additional objectclasses
var OCs = new Array( "favoriteColorOC" );
if( cd.addUser( "uid=joe, ou=People, o=acme.com", attrs, OCs ) == false
)
{
java.lang.System.out.println( "Problems adding user entry joe"
);
}

```

deleteUserByCN(userCN)

Deletes the user entry specified by the Common Name (CN) *userCN* from the corporate directory. Returns `true` if the deletion is successful. Returns `false` if the deletion failed.

Parameter	Data Type	Description
userCN	String	CN deleted by the user entry.

Example:

```

var cd = getCorporateDirectory();
var joe = cd.getUserById( "joe" );
if( cd.deleteUserByCN( joe.cn ) )
    java.lang.System.out.println( "User entry joe deleted" );
else
    java.lang.System.out.println( "Problems deleting user entry joe" );

```

deleteUserByDN(userDN)

Deletes the user entry specified by the Distinguished Name (DN) *userDN* from the corporate directory. Returns `true` if the deletion is successful. Returns `false` if the deletion failed.

Parameter	Data Type	Description
<code>userDN</code>	String	DN deleted by the user entry.

Example:

```
var cd = getCorporateDirectory();
var joe = cd.getUserById( "joe" );
if( cd.deleteUserByDN( joe.getDN() ) )
    java.lang.System.out.println( "User entry joe deleted" );
else
    java.lang.System.out.println( "Problem deleting user entry joe" );
```

deleteUserById(uid)

Deletes the user entry specified by `userId` from the corporate directory. Returns `true` if the deletion is successful. Returns `false` if the deletion failed.

Parameter	Data Type	Description
<code>userDN</code>	String	DN deleted by the user entry.

Example:

```
var cd = getCorporateDirectory();
var joe = cd.getUserById( "joe" );
if( cd.deleteUserByCN( joe.getUserId() ) )
    java.lang.System.out.println( "User entry joe deleted" );
else
    java.lang.System.out.println( "Problem deleting user entry joe" );
```

getUserByCN(userCN)

Returns the user entry corresponding to the Common Name (CN) *userCN* as a JavaScript User object. User object is a PAE-specific object. Returns null if no user entry corresponds to *userDN*.

Parameter	Data Type	Description
userCN	String	CN retrieved by the user entry.

Example:

```
var cd = getCorporateDirectory();
var joe = cd.getUserByCN( "Joe Cool" );
if( joe == null )
    java.lang.System.out.println( "Cannot find entry for user joe" );
else
    java.lang.System.out.println( "Found user joe" );
```

getUserByDN (userDN)

Returns the user entry corresponding to the Distinguished Name (DN) *userDN* as a JavaScript User object. User object is a PAE-specific object. Returns null if no user entry corresponds to *userDN*.

Parameter	Data Type	Description
userDN	String	DN retrieved by the user entry.

Example:

```
var cd = getCorporateDirectory();
var joe = cd.getUserByDN( "uid=joe, ou=People, o=acme.com" );
if( joe == null )
    java.lang.System.out.println( "Cannot find entry for user joe" );
else
    java.lang.System.out.println( "Found user joe" );
```

getUserById (uid)

Returns the user entry corresponding to the user id *userId* as a JavaScript User object. The User object is a PAE-specific object. Returns null if no user entry corresponds to *userId*.

Parameter	Data Type	Description
uid	String	User ID retrieved by the user entry.

Example:

```
var cd = getCorporateDirectory();
var joe = cd.getUserById( "joe" );
if( joe == null )
    java.lang.System.out.println( "Cannot find entry for user joe" );
else
    java.lang.System.out.println( "Found user joe" );
```

modifyUserByCN (userCN, attrName, attrValue, operation)

See modifyUserByDN (userDN, attrName, attrValue, operation).

Parameter	Data Type	Description
userCN	String	CN modified by the user entry.
attrName	String	Name of the user entry attribute to modify.
attrValue	String	Value of the user entry attribute to modify.
operation	String	Modification operation performed.

Example:

```
var cd = getCorporateDirectory();
if(cd.modifyUserByCN( "uid=joe, ou=People, o=acme.com",
"favoriteColor", "blue", "ADD" ) == false )
{
    java.lang.System.out.println( "Failed to modify user joe"
);
};
```



```

}
else if( cd.modifyUserByCN( "uid=joe, ou=People, o=acme.com",
"favoriteColor", "red", "REPLACE" ) == false )
{
    java.lang.System.out.println( "Failed to modify user joe");
}
else if( cd.modifyUserByCN( "e, ou=People, o=acme.com", "favoriteColor",
"", "DELETE" ) == false )
{
    java.lang.System.out.println( "Failed to modify user joe");
}

```

modifyUserByDN (userDN, attrName, attrValue, operation)

Modifies the user entry specified by the Distinguished Name (DN) *userDN* in the corporate directory. Returns true if the modification is successful. Returns false if the modification fails.

Parameter	Data Type	Description
userDN	String	DN modified by the user entry.
attrName	String	Name of the user entry attribute to modify.
attrValue	String	Value of the user entry attribute to modify.
operation	String	Modification operation performed.

The type of modification is specified by the operation, which can have the following values:

ADD: adds the attribute attrName with the value attrValue to the user entry. If attrName has multiple values, the operation can result in multiple instances of attrName in the user entry. attrName should be a valid attribute of one of the object classes with which the user entry complies.

REPLACE: replaces the value currently associated with attrName with the new value attrValue.

DELETE: deletes attrName from the user entry.

Example:

```

var cd = getCorporateDirectory();
if(cd.modifyUserByDN( "uid=joe, ou=People, o=acme.com", "favoriteColor", "blue",
"ADD" ) == false )
{
    java.lang.System.out.println( "Failed to modify user joe"
);
}
else if( cd.modifyUserByDN( "uid=joe, ou=People, o=acme.com", "favoriteColor",
"red", "REPLACE" ) == false )
{
    java.lang.System.out.println( "Failed to modify user joe");
}
else if( cd.modifyUserByDN( "e, ou=People, o=acme.com", "favoriteColor", "",
"DELETE" ) == false )
{
    java.lang.System.out.println( "Failed to modify user joe");
}

```

modifyUserById (userID)

See `modifyUserByDN (userDN, attrName, attrValue, operation)`.

Parameter	Data Type	Description
userID	String	User ID modified by the user entry.

Example:

See `modifyUserByDN (userDN, attrName, attrValue, operation)`. You must pass in a user ID rather than a DN.

User

The `User` object has publically accessible methods and exposes the user attributes making up an LDAP user entry. You can treat the `User` object as a hashtable of attribute key/value pairs where the keys are the attribute names as they appear in the LDAP entry. In the following code sample, you can also access the attribute `favoritecolor`, which is an attribute available from the user `joe`'s entry.

Example:

```
var cd = getCorporateDirectory();
var u = cd.getUserById( "joe" );
java.lang.System.out.println( "joe's email is " + u.mail );
java.lang.System.out.println( "Joe's favorite color is " +
u.favoritecolor );
```

The following methods can be called on the `User` object:

- `getUserId`
- `getDN`

getUserId()

Returns the user ID for the current user. The user ID attribute can also be accessed directly as the `id` property of the user object.

Example:

```
var cd = getCorporateDirectory();
var u = cd.getUserByDN( "uid=joe, ou=People, o=acme.com" );
java.lang.System.out.println( "Joe's user id is " + u.getUserID() );
var pi = getProcessInstance();
u = pi.getCreatorUser();
java.lang.System.out.println( "Creator's user id is " + u.getUserId() );
```

getDN()

Returns the Distinguished Name (DN) for the current user. The user DN attribute can also be accessed directly as the `dn` property of the user object.

Example:

```
var cd = getCorporateDirectory();
var u = cd.getUserById( "joe" );
java.lang.System.out.println( "Joe's user DN is " + u.getDN() );
var pi = getProcessInstance();
u = pi.getCreatorUser();
java.lang.System.out.println( "Creator's user DN is " + u.getDN() );
```

Logging and Error Handling Global Functions

The following global functions are available for adding entries to error, informational, and history logs. Note that the two forms, `Msg` and `Message`, are identical. The duplicate names are provided for convenience only.

- `logErrorMsg`
- `logHistoryMsg`
- `logInfoMsg`
- `logSecurityMsg`
- `setErrorMsg`
- `logErrorMessage`
- `logHistoryMessage`
- `logInfoMessage`
- `logSecurityMessage`
- `setErrorMessage`

`logErrorMsg (label, context)`

Adds an entry in the error log of the application. This information is displayed to the administrator when viewing the error log.

Parameter	Data Type	Description
<code>label</code>	String	Label for the error log's entry.
<code>context</code>	Object	(Optional) Hashtable of key/value pairs to print in the log entry.

The following code could be used in a completion script or automation script to log an error message when a user tries to submit a document for review when the document is too long.

Example:

```
// LOG AN ERROR MESSAGE
var pi = getProcessInstance();
var author = pi.getData("author");
var docName = pi.getData("docName");
var pageCount = parseInt (pi.getData("pageCount"));

if (pageCount > 5000) {
    var error1 = new Object ();
    error1.author = author;
    error1.docName = docName;
```

```

    error1.pageCount = pageCount;
    logErrorMsg ("PAGE_COUNT_TOO_LONG", error1);
}

```

logHistoryMsg (label, comment)

Adds a row in the history log. This information is displayed to users when viewing Details & History in a work item list.

Parameter	Data Type	Description
label	String	Label for the entry in the history log.
comment	String	(Optional) Additional comment for the history event.

The *label* is a string to be used as the label for the entry in the history log, and the optional argument *comment* is a string describing the entry.

Use this function to add information for the users about the work items. The following code could be used in a completion script to add details when an author submits a document for review.

Example:

```

// LOG A HISTORY MESSAGE
var pi = getProcessInstance();
var author = pi.getData("author");
var docName = pi.getData("docName");
var pageCount = parseInt (pi.getData("pageCount"));
logHistoryMsg ("Doc Submitted for Review",
    " \nAuthor: " + author +
    " \nDocument name: " + docName +
    " \nPage count: " + pageCount);

```

logInfoMsg(label, context)

Adds an entry in the info log of the application. This information is displayed to the administrator when they view the info log.

Parameter	Data Type	Description
-----------	-----------	-------------

label	String	Label for the entry in the history log.
context	String	(Optional) Hashtable of key/value pairs to print in the log entry.

The *label* is a string to be used as the label for the entry in the info log, and the optional argument *context* is an object that has a variable for each property of the info message.

Use this function to show information about the execution of scripts. This is particularly useful for viewing the progress of script execution while debugging scripts. The following code could be used in a completion or automation script to log information about a document that has been submitted for review.

Example:

```
// LOG AN INFO MESSAGE
var pi = getProcessInstance();
var author = pi.getData("author");
var docName = pi.getData("docName");
var pageCount = parseInt (pi.getData("pageCount"));

var info1 = new Object ();
info1.author = author;
info1.docName = docName;
logInfoMsg ("DOC_SUBMITTED_FOR_REVIEW", info1);
```

logSecurityMsg (label, context)

Adds an entry in the security log of the application. This information is displayed to the administrator when they view the security log.

Parameter	Data Type	Description
label	String	Label for the entry in the security log.
context	String	(Optional) Hashtable of key/value pairs to print in the log entry.

The *label* is a string to be used as the label for entry in the security log, and the optional argument *context* is an object that has a variable for each property of the error message.

The following code could be used in a completion script or automation script to log a warning when an unauthorized user tries to perform a task that accesses an external database.

Example:

```
// LOG A SECURITY MESSAGE
var pi = getProcessInstance();
var username = pi.getData("username");
var password = pi.getData("password");
var externalDB = pi.getData("externalDBName");

var securityError1 = new Object ();
securityError1.username = username;
securityError1.password = password;
securityError1.externalDB = externalDB;
logSecurityMsg ("EXTERNAL_DB_ACCESS_NOT_ALLOWED", securityError1);
```

setErrorMsg (errMessage)

This function enables the designer to add additional information that describes why a script failed. This information is displayed to the participant as part of the error dialog box.

Parameter	Data Type	Description
errMessage	String	Added to the error dialog box when an error occurs during processing.

The following code could be used in a completion script to add a message explaining that the completion script failed because the submitted document has too many pages.

Example:

```
function checkPageCount () {
// ADD A COMPLETION SCRIPT ERROR MESSAGE
var pi = getProcessInstance();
var docName = pi.getData("docName");
var pageCount = parseInt (pi.getData("pageCount"));

if (pageCount > 5000) {
    setErrorMsg ("The page count for " + docName +
```

```

    " is " + pageCount +
    " which exceeds the maximum page count allowed. " +
    " Did you really write such a long document?");
return false;
}
else return true;
}

```

Assignment, Completion, and Email Scripts

This section lists the predefined assignment, completion, and email scripts available in Process Automation Edition. These predefined scripts (which are also global functions) can be called from other scripts. For example, an expiration handler script could call the predefined function `toUserById()`. The value returned by `toUserById()` could in turn be passed to the `assignTo()` method on a work item. The `assignTo()` method would then re-assign the work item to a user with a given user ID.

checkParallelApproval (dataField, stopAction)

This function is meant for use as a completion script for work items that use the `toParallelApproval()` assignment script. The *dataField* must be the name of the data field that was specified in `toParallelApproval()`, and is used to keep track of who has completed the work item so far. The *stopAction* is the name of the action that a user can take to stop the approval process.

Parameter	Data Type	Description
<code>dataField</code>	String	Name of a data field.
<code>stopAction</code>	String	Name of an action.

If an activity (work item) uses the `toParallelApproval()` assignment script, the activity should have two possible actions for the assignees to take when they complete the work item. One should indicate approval, and the other should indicate disapproval. The activity should also use the `checkParallelApproval()` completion script, which performs the

“disapproval” action as soon as any one assignee selects it, the theory being that it takes everyone to approve the work item, but only one dissenter to disapprove it.

Example:

```
checkParallelApproval( "trackerField", "Reject" );
```

defaultNotificationHeader()

Returns the default notification header for a notification message body. The header contains information about the current work item, such as the current activity name, the process instance ID, the creation date of the process instance and the expiration date (if any).

This function may be used as the notification body script by itself, or may be embedded in your own notification body script. You may also use this function from a template evaluated using `evaluateTemplate()`. The function may only be used successfully from a script associated with a notification; if used anywhere else, an empty string is returned.

The text returned from this function will depend upon the content-type of the notification. If the content-type is `text/html`, the header will be a series of HTML tags; if the content-type is `text/plain`, the header will be plain text.

Example:

```
function emailBody( )
{
  var body = "Email message body <br>";
  body += defaultNotificationHeader();
  return body;
}
```

defaultNotificationSubject()

Returns the default notification subject for the notification subject line. The subject contains information about the current process instance, such as the process instance ID the priority and the title string.

This function may be used as the notification subject script by itself, or may be embedded in your own notification subject script. The function may only be used successfully from a script associated with a notification; if used anywhere else, an empty string is returned.

Example:

```
function emailSubject( )
{
var subject = "Subject: " + defaultNotificationSubject();
return subject;
}
```

emailByDN(DN)

Returns a string of comma-delimited email addresses for the user with the given Distinguished Name (DN). The `mail` attribute for the user must contain a valid email address in the corporate user directory. If the `mail` attribute of the user does not have a value, this function logs an error and returns `null`. This function is intended for use as a notification script, but can be used anywhere that a string of email addresses is needed.

Parameter	Data Type	Description
<code>userDN</code>	String	DN of the user whose email address is returned.

Example:

```
function getEmailByDN( )
var cd = getCorporateDirectory();
var u = cd.getUserById( "joe" );
return emailByDN( u.getDN() );
}
```

emailById(userId)

Returns a string of comma-delimited email addresses for the user with the given user ID. The `mail` attribute for the user must contain a valid email address in the corporate user directory. If the `mail` attribute of the user does not have a value, this function logs an error and returns `null`. This function is intended for use as a notification script, but can be used anywhere that a string of email addresses is needed.

Parameter	Data Type	Description
<code>userId</code>	String	User whose email address is returned.

Example:

```
emailById( "joe" );
```

emailOfAssignees()

Returns a string of comma-delimited email addresses for all the assignees of the work item. The `mail` attribute for each assignee must contain a valid email address in the corporate user directory. If the `mail` attribute is empty for any assignee, no address is added to the string for that assignee. If no assignee has a value in their `mail` attribute, the function logs an error message and returns `null`. This function is intended for use as a notification script, but can be used anywhere that a string of email addresses is needed.

Example:

```
emailOfAssignees();
```

emailOfCreator()

Returns a string of the email address of the user who created the process instance. The user's `mail` attribute must contain a valid email address in the corporate user directory. If the `mail` attribute of the user does not have a value, this function logs an error and returns `null`. This function is intended for use as a notification script, but can be used anywhere that a string of email addresses is needed.

Example:

```
var pi = getProcessInstance();
var creatorEmail = emailOfCreator();
pi.setData( "creatorEmail", creatorEmail );
```

emailOfRole(roleName)

Returns a string of the email address of the user performing the given role. The user's `mail` attribute must contain a valid email address in the corporate user directory. If the `mail` attribute of the user does not have a value, this function logs an error and returns `null`. This function is intended for use as a notification script, but can be used anywhere that a string of email addresses is needed.

Parameter	Data Type	Description
roleName	String	Role whose email address is returned.

Example:

```
emailOfRole( "reviewer" );
```

randomToGroup(groupName)

This function returns an array containing a Distinguished Name (DN) of one member of the group. The group member is selected at random. This function is intended to be used as an assignment script but can be used anywhere that an array of DNs is needed.

Parameter	Data Type	Description
groupName	String	Name of a group.

Example:

```
var userDNArray = randomToGroup( "helpDesk" );
java.lang.System.out.println( "The DN of a helpdesk user is: " +
userDNArray[ 0 ] );
```

toCreator()

Returns a JavaScript array with the Distinguished Name (DN) of the user who created the process instance. This method is the default assignment script assigned to user activities when a node is created on the Process Map.

Example:

```
var creatorDNArray = toCreator();
java.lang.System.out.println( "Creator's DN is " + craetorDNArray[ 0 ]
);
```

toGroup(groupName)

Returns a JavaScript array of Distinguished Names (DNs) of all members of the specified group. If the parameter *groupName* does not correspond to an actual group in the application's Groups and Roles folder, returns `null`.

Parameter	Data Type	Description
<code>groupName</code>	String	Name of the group.

Example:

```
var groupDNs = toGroup( "reviewers" );
if( groupDNs != null )
{
    java.lang.System.out.println( "Number of members in the group" +
groupDNs.length );
    for (var i = 0; i <= groupDNs.length; i++)
    {
        java.lang.System.out.println( "Member #" + i + ": " + groupDNs[i ]
);
    }
}
else
    java.lang.System.out.println( "No such group reviewers" );
```

toManagerOf (userId)

This function returns an array containing the Distinguished Name (DN) of the manager of the user with the given user ID. The `manager` attribute of the given user must contain a DN in the corporate user directory. This function is intended to be used as an assignment script but can be used anywhere that an array of DN's is needed.

Parameter	Data Type	Description
<code>userId</code>	String	A user ID.

Example:

```
var managerDNArray = toManagerOf( "sijacic" );
java.lang.System.out.println( "Manager of user sijacic is: " +
```

```
managerDNArray[0] );
```

toManagerOfCreator()

This function returns an array containing the Distinguished Name (DN) of the manager of the creator of the process instance. The `manager` attribute of the creator user must contain a DN in the corporate user directory. This function is intended to be used as an assignment script but can be used anywhere that an array of DNs is needed.

Example:

```
var managerDNArray = toManagerOfCreator();
java.lang.System.out.println( "Manager of the process creator is: " +
ManagerDNArray[ 0 ] );
```

toManagerOfRole(role)

This function returns an array containing the Distinguished Name (DN) of the manager of the user defined as the given role. The `manager` attribute of the user fulfilling the role must contain a DN in the corporate user directory. This function is intended to be used as an assignment script but can be used anywhere that an array of DNs is needed.

Parameter	Data Type	Description
<code>role</code>	String	Name of the role.

Example:

```
var managerDNArray = toManagerOfRole( "reviewer" );
java.lang.System.out.println( "Manager of the reviewer is: " +
managerDNArray[ 0 ] );
```

toParallelApproval(arrayOfUserDNs, dataField)

This function should be used as an assignment script to assign a work item to several users who must all complete the work item. The `arrayOfUserDNs` argument is an array of distinguished names, and `dataField` is a data field that keeps track of who has performed the work item and who still needs to do it. This field is a computed field of length 2000 that must add to the data dictionary.

Parameter	Data Type	Description
arrayOfUserDNs	String	Name of the role.
dataField	String	Name of a data field in the current process.

If an activity (work item) uses the `toParallelApproval()` assignment script, the activity should have two possible actions for the assignees to take when they complete the work item. One action should indicate approval, and the other should indicate disapproval. The activity should also use the `checkParallelApproval()` completion script, which performs the “disapproval” action as soon as any one assignee selects it. The idea is that it takes everyone to approve the work item, but only one dissenter to disapprove it.

Example:

```
var cd = getCorporateDirectory();
var user1 = cd.getUserById( "user1" );
var user2 = cd.getUserById( "user2" );
var user3 = cd.getUserById( "user3" );
var approvers = new Array( user1.getDN(), user2.getDN(), user3.getDN() );
toParallelApproval( approvers, "trackerField" );
toParallelApproval(toGroup( "approvers" ), "trackerField" );
```

toUserById(userId)

This function returns an array containing the Distinguished Name (DN) of the user with the given user ID. This function is intended to be used as an assignment script but can be used anywhere that an array of DNs is needed.

Parameter	Data Type	Description
userId	String	A user ID.

Example:

```
var userDNArray = toUserById( "sijacic" );
java.lang.System.out.println( "The DN of the user is: " + userDNArray[ 0 ] );
```

toUserFromField(dataField)

This function returns an array containing the Distinguished Name (DN) of the user whose user ID is the value of the given data field. This function is intended to be used as an assignment script but can be used anywhere that an array of DNs is needed.

Parameter	Data Type	Description
dataField	String	Name of a data field.

Example:

```
var userDNArray = toUserFromField( "approver" );
java.lang.System.out.println( "The DN of the user is: " + userDNArray[ 0
] );
```

Miscellaneous Global Functions

This section summarizes the global functions that are neither log and error functions nor predefined scripts.

checkUserDNs(arrayOfUserDNs)

Verifies that all the DNs defined in the *arrayOfUserDNs* are valid and exist in the corporate directory. Return *true* if all the DNs are valid (or if the array is empty) otherwise returns *false*.

Parameter	Data Type	Description
arrayOfUserDNs	Array	List of DNs.

Example:

```
var dnArray = new Array( "uid=sijacic, o=mcom.com",
    "uid=atam, o=mcom.com",
    "uid=souvik, o=mcom.com",
    "uid=michal, o=mcom.com",
    "uid=raghavan, o=mcom.com" );
```



```
// Check to see if all of the DNs are valid
if( checkUserDNs( dnArray ) )
java.lang.System.out.println( "All DNs are valid" );
else
java.lang.System.out.println( "One of the DNs is not valid" );
```

ejbLookup(jndiName)

Looks up an EJB that can be found under the given JNDI name. If no EJB exists under this name, an EJB exception, `javax.naming.NamingException`, is thrown. PAE handles this exception by moving the process instance into the designated Exception Manager node.

If the bean can be found under the given JNDI name, the home interface of that bean is returned. The home interface is used to create new instance of the EJB or find existing instances of the given EJB. The definition of the home interface is completely application-dependent; it typically consists of several `create()` and `finder()` methods. This function call is equivalent to the `getJndiNamingContext().lookup(jndiName)` function call.

Parameter	Data Type	Description
<code>jndiName</code>	String	JNDI name of the EJB to look up.

Example:

```
var home = ejbLookup("Netscape/CreditCardServer");
// Create a credit card server that can do card authorizations.
var creditCardServer = home.create ( "mastercard" );
```

evaluateTemplate(templateName)

Used primarily in email notifications, but can be used for any template-like evaluation.

For email notifications, use `evaluateTemplate()` in the email body. When you create an email notification in Process Builder, specify `evaluateTemplate("templateFile")` as the body script, where `templateFile` is the name of the template file relative to the application directory on the server. From Process Builder, you should include your

template files in a folder called `templates`, a directory located in the application directory of Process Builder. When you deploy, the same directory structure is replicated on the server.

`evaluateTemplate()` also evaluates JavaScript segments contained in the template file. These segments are identified by `<script language="Rhino">` tags.

In an HTML page containing client-side JavaScript, you can replace

```
<script language="JavaScript">
```

with

```
<script language="Rhino">
```

The PAE engine will evaluate the JavaScript in the same manner the browser does.

You can access the entire engine API (e.g., `getProcessInstance()`, `getWorkItem()`, etc.) from within these segments. To send output to an HTML file, use `document.write()` with the output you want to send provided as an argument.

`evaluateTemplate()` returns a string. The string is made up of the contents of the template file with the Rhino segments evaluated. If there are no Rhino segments in the file, the string simply contains the file contents. If there are Rhino segments, the string contains the evaluated JavaScript rather than the Rhino segments.

Example:

```
var fileContents = evaluateTemplate( "templateFile" );
var cs = getContentStore();
cs.upload( fileContents, url );
```

expireIn(val, unit)

Returns a JavaScript `Date` object set to the absolute `datetime` when the current work item will expire.

Parameter	Data Type	Description
-----------	-----------	-------------

val	Integer	Numeric value specifying the number of units in which to expire the workitem.
unit	String	Specifies the unit of time measurement value represents.

This function is normally used to specify the expiration time for a particular activity, and is usually set by the designer through Process Builder. In the event that you choose to specify the arguments for this function manually, the parameter unit can contain the following values:

- minutes
- hours
- days
- weeks
- months

Note that a month is presumed to contain 30 days. Asking a work item to expire one month from the date January 1st results in an expiration date of January 30th, not February 1st.

Example:

```
expireIn( 6, "minutes" );    // expire in 6 minutes
expireIn( 30, "days" );    // expire in 30 days
expireIn( 1, "month" );    // expire in 30 days
```

getAction()

Returns the name of the action (the button name) used to complete the current activity. The action name corresponds to the name of the button clicked when the work item form is submitted. This function can be used in any script, but is best used from the completion script of a user activity.

Example:

```
function completion( )
{
var pi = getProcessInstance();
pi.setData( "submittedAction", getAction() );
```

```
return true;
}
```

getApplicationName()

Returns the name of the application (for example: "wfDataSheet") as a string.

Example:

```
var appName = getApplicationName();
java.lang.System.out.println( "The application name is: " + appName );
```

getApplicationPath()

Returns the pathname of the directory containing the files for this application as a string.

Example:

```
var appPath = getApplicationPath();
java.lang.System.out.println( "The application path is: " + appPath );
```

getApplicationPrettyName()

Returns the pretty name of the application (for example: "DataSheet Management") as a string.

Example:

```
var appPN = getApplicationPrettyName();
java.lang.System.out.println( "The application pretty name is: " + appPN );
```

getBaseForFileName (processId)

Given a unique process instance ID, this function returns a string of the base file name for the content store for that process instance.

Parameter	Data Type	Description
processId	String	A process ID.

The following code creates a path name for the file `myFile.html` in the content store. Note that this code does NOT create the file, it simply creates a string that can be used as the file name by methods on a `contentStore` object, such as `move()` and `store()`.

Example:

```
var pi = getProcessInstance();
var pid = pi.getInstanceId();
var contentStorePath = getBaseForFileName (pid);
var newFileName = contentStorePath + "myFile.html";
```

getConnector(connectorKey)

Given a connector key, returns a connector object from the list of connector objects available for use by scripts.

Parameter	Data Type	Description
connectorKey	String	A connector ID (previously set by <code>setConnector()</code>).

Example:

```
function initialisationScript( )
{
    var c = new Packages.com.acme.DBConnection();
    setConnector( "dbConnector" );
    return true;
}
var c = getConnector( "dbConnector" );
```

getContentStore ()

Returns a `contentStore` object connected to the content store. See the section `ContentStore` for details of the methods on this object.

Example:

```
var cs = getContentStore();
```

getContentStore(httpURL,user,password)

General purpose content store method. This method enables you to authenticate to arbitrary HTTP servers for file manipulation. You must have a content store object to use any of the content store methods.

Parameter	Data Type	Description
httpURL	String	URL of an HTTP server capable of acting as a content store.
user	String	User ID of the user who will authenticate the content store specified by httpURL.
password	String	Password for the specified user.

Example:

```
var publishCS = getContentStore(
    "http://publish.netscape.com/CS", "sijacic", "password");
```

getCreatorUserId ()

Returns the user ID for the creator user of the current process instance.

This function is intended to be used as the `initiate-as` script for sub-process nodes. When the child process instance is created, its creator must be determined. Normally, the creator of the child process instance is the same user that authenticated with the server at the time the sub-process node was executed. However, it may be desirable (or necessary) to set the child creator to the same user as the creator of the parent process instance.

If you choose to use this function to change the child creator user, the authenticating (or current) user must be a member of the trusted users group in the child application.

Example:

```
var pi = getProcessInstance();
var creatorUser = pi.getCreatorUser();
var creatorUserId = creatorUser.getUserId();
```

getCorporateDirectory ()

Returns a `corporateDirectory` object connected to the corporate directory in the Directory Server. See the section `CorporateDirectory` for details of the methods on this object.

Example:

```
var corpDir = getCorporateDirectory();
```

__getIncludePath ()

Returns the current inclusion path used by the server to search for JavaScript source files. This search path resembles in behavior the `PATH` mechanism used in Windows NT and UNIX to search for executables and libraries when a file name is not absolute.

The directory `installDir/resources/server/js` (where `installDir` is PAE installation directory on the server machine) is included in the search path by default.

Example:

```
// Returns the current search path
var includePath = __getIncludePath();

// Add our own directory to the search path
includePath += ';' + "d:\\tmp";

// Set the search path
__setIncludePath( includePath );
```

getJndiNamingContext()

Returns the current NAS JNDI naming context used to look up the home interface of an EJB.

Use this function to access EJBs from within a Custom Activity. You can pass the instance of the naming context to the Custom Activity and use the `lookup()` method on this context to obtain home interfaces of the beans.

Example:

```
var namingContext = getJndiNamingContext();
```

getProcessInstance ()

Returns a JavaScript ProcessInstance object associated with the current work item.

See the section ProcessInstance for details of the methods on this object.

Example:

```
var wi = getWorkItem();
var pi = getProcessInstance();
java.lang.System.out.println( "The Process Instance ID: " +
pi.getInstanceId() );
```

getSubProcessInstance ()

Returns a ProcessInstance object corresponding to the completed child process instance. This function should only be used from the completion script of a sub-process node. The object returned by this function has all the same methods expected of a ProcessInstance object, except that the information contained within corresponds to the completed child process instance instead of the parent. If this function is called from any location other than the completion script of a sub-process node, null is returned.

Example:

```
// Completion script of sub-process node
function parentCompletion( )
{
var pi = getProcessInstance();          // parent process instance
var spi = getSubProcessInstance();     // child process instance

// Map some of the data elements from the child back to the
// parent process instance
pi.setData( "childStatus", spi.getData( "status" ) );
pi.setData( "numApprovers", spi.getData( "numApprovers" ) );
return true;
}
```

getWorkItem ()

Returns a workItem object that represents the current work item. See the section WorkItem for details of the methods on this object.

Example:

```
var wi = getWorkItem();
```

__includeFile (fileName)

Reads in the contents of a JavaScript source file and evaluates the contents. Any JavaScript functions and objects defined within the file are available to all user scripts.

`__includeFile()` and its “helper” functions, `__getIncludePath()` and `__setIncludePath()`, are used by the server to define the built-in JavaScript functions available to user scripts.

If you want to access JavaScript functions or objects from user scripts and do not want to include the script files with every application you develop, you can externalize the functions in a separate JavaScript file and include the functions in the server’s global run-time scope using `__includeFile()`.

Parameter	Data Type	Description
fileName	String	Pathname of the JavaScript file to evaluate.

Example 1:

```
// Sample JavaScript file

function checkCreator( userId )
{
  var pi = getProcessInstance();
  var u = pi.getCreatorUser();
  // If the process instance creator matches the
  // userId, return true, else false.
  if( userId == u.getUserId() )
    return true;
  else
    return false;
}
```

Example 2

```
// Sample includeList.js file

var myIncludePath = "d:\\tmp";
var includePath = __getIncludePath() + ';' + myIncludePath;
```

```
__setIncludePath( includePath );
__includeFile( "myFunctions.js" );
```

mapTo(fieldName)

Used only from a custom activity as a means of mapping values from the output hashtable back to the process instance. The *fieldName* parameter specifies the name of a data element in the process instance. This data element receives the value of the current output parameter. When the custom activity's `perform()` call is complete, the server iterates through the hashtable, one element at a time, and attempts to map the element's value to a field in the process instance. Returns `true` if data mapping succeeds. Returns `false` if a problem has occurred.

Parameter	Data Type	Description
<code>fieldName</code>	String	Name of a data element in the process instance.

Example:

```
mapTo( "customerName" );
```

mount(jndiName)

Looks up and creates an instance of a stateless session bean, or EJB, found under the given JNDI name. If no bean exists under this name, a `javax.naming.NamingException` is thrown. PAE handles this exception by moving the process instance into the designated Exception Manager node. If the EJB can be found under the given JNDI name, an instance of it is created by calling the `create()` method on the returned home interface of the bean. This function call is equivalent to the `ejbLookup(jndiName). create()` function call.

Parameter	Data Type	Description
<code>jndiName</code>	String	JNDI name of the EJB to "mount."

Example:

```
var creditServer = mount("Netscape/CreditCardServer");
```

setConnector(connKey, connObject)

Adds a connector object (such as a database connection) indexed by key to the list of connector objects that can be used by scripts.

Parameter	Data Type	Description
connKey	String	Key used to retrieve the associated object (through <code>getConnector()</code>).
connObject	Object	Object stored and associated with the specified key.

JavaScript objects saved inside the Connector, such as those created using `var obj = new Object()`, are converted into Java objects based on the mapping table below before being stored in the connector

JavaScript Data Type	Converted Java Data Type
string	java.lang.String
number	java.lang.Double
boolean	java.lang.Boolean
date	java.util.Date
array	java.lang.Object[]
object	java.util.Hashtable
function	java.lang.String (decompiled source)
script	java.lang.String (decompiled source)
java.lang.Object (wrapped)	java.lang.Object (unwrapped)
java.lang.Object	java.lang.Object (pass-through)

Example:

```
function initialisationScript( )
{
var c = new Packages.com.acme.DBConnection();
setConnector( "dbConnector", c );
```

```

return true;
}

// This DB connection can now be used in any script.
var c = getConnector( "dbConnector" );

```

__setIncludePath (includePath)

Allows the user to specify the path the server uses to search for JavaScript source files. The path is a delimited list of absolute directory path names. The delimiter depends upon the system:

- : (a colon) for UNIX
- ; (a semicolon) for Windows

Searches start from the first directory and proceed to the last directory until the file is located. The directory *installDir/resources/server/js* (where *installDir* is where PAE is installed on the server machine) is included in the search path by default. However, once you set your own path using this function, the current search path is overwritten. Thus, calls to `__setIncludePath` should specify the entire search path by appending the new search directory to the current search path.

Parameter	Data Type	Description
includePath	String	Ssearch path the server uses to locate JavaScript source files.

Example:

```

var includePath = __getIncludePath();
includePath += ';' + "d:\\tmp";

// Set the search path. Note how the variable includePath also
// specifies the existing search path. Every call to __setIncludePath
// overwrites the previous search path.
__setIncludePath( includePath );

```

setRedirectionURL(stringURL)

Redirects the participant's browser to the specified location in `stringURL`.

Invoke this method from a completion script. `setRedirectionURL()` is typically used for multiple-screen data entry where the same person may enter data across several screens or user activities. The process subsequently becomes a wizard-like forms entry screen.

Parameter	Data Type	Description
<code>stringURL</code>	String	A valid URL.

Example:

```
function scriptComplete( )
{
// Redirect to the Netscape Home Page
setRedirectionURL( "http://home.netscape.com" );
// This will cause the same process instance to display after the
// user clicks an action button
setRedirectionURL( url_OnDisplayProcessInstance() );

// Don't forget to return true; this is still a completion script
return true;
}
```

`url_OnDisplayHistory()`

Returns a string containing the URL that points to the history list for the current process instance. This function is usually used in conjunction with `setRedirectionURL()` to redirect a user to the history page of the current process instance.

Example:

```
function scriptComplete( ) {
// Redirect to the History Page of this process instance
setRedirectionURL( url_OnDisplayHistory() );
}
```

`url_OnDisplayProcessInstance()`

Returns a string containing the URL that points the current process instance. This function is usually used in conjunction with `setRedirectionURL()` to redirect a user to the current process instance.

Example:

```
function scriptComplete( )
{
// Redirect to the current process instance
setRedirectionURL( url_OnDisplayProcessInstance() );
}
```

url_OnDisplayWorklist()

Returns a string containing the URL that points to the user's work list. This function is usually used in conjunction with `setRedirectionURL()` to redirect a user to their work list.

Example:

```
function scriptComplete( )
{
// Redirect to the user to their work list
setRedirectionURL( url_OnDisplayWorklist() );
}
```

url_OnListApplications()

Returns a string containing the URL listing applications installed on the PAE cluster. This function is usually used in conjunction with `setRedirectionURL()` to redirect a user to the list of applications installed on the cluster.

Example:

```
function scriptComplete( )
{
// Redirect to the list of applications on this cluster
setRedirectionURL( url_OnListApplications() );
}
```

url_OnListEntryNodes()

Returns a string containing the URL that points to the list of entry points in the current application. This function is usually used in conjunction with `setRedirectionURL()` to redirect a user to the list of entry points in the current application.

Example:

```
function scriptComplete( )
{
// Redirect to the list of entry points of the current application
setRedirectionURL( url_OnListEntryNodes() );
}
```

Alphabetical Summary of JavaScript Objects

The following table lists the JavaScript methods and functions available in Process Automation Edition (PAE):

JavaScript method or function	Where Used in PAE
__getIncludePath	global function
__includeFile	global function
__setIncludePath	global function
addUser	corporateDirectory
assignTo	workItem
checkParallelApproval	predefined script
checkUserDNs	global function
deleteUserByCN	corporateDirectory
deleteUserByDN	corporateDirectory
deleteUserById	corporateDirectory
download	contentStore
emailByDN	predefined script
emailById	predefined script
emailOfAssignees	predefined script
emailOfCreator	predefined script
emailOfRole	predefined script
evaluateTemplate	global function
exists	contentStore

JavaScript method or function	Where Used in PAE
expire	workItem
expireIn	global function
extend	workItem
getApplicationName	global function
getApplicationPath	global function
getApplicationPrettyName	global function
getAssigneesDN	workItem
getBaseForFileName	global function
getConclusion	global function
getConnector	global function
getContent	contentStore
getContentStore	global function
getCorporateDirectory	global function
getCreationDate	processInstance
getCreationDate	workItem
getCreatorDN	processInstance
getCreatorUser	processInstance
getCreatorUserId	global function
getCurrentActivityCN	workItem
getData	processInstance
getEntryNodeName	processInstance
getException	contentStore
getExitNodeName	processInstance
getExpirationDate	workItem
getInstanceId	processInstance
getPriority	processInstance
getProcessInstance	global function
getRoleDN	processInstance

JavaScript method or function	Where Used in PAE
getRoleUser	processInstance
getRootURL	contentStore
getStatus	contentStore
getSubProcessInstance	global function
getTitle	processInstance
getUserByCN	corporateDirectory
getUserByDN	corporateDirectory
getUserById	corporateDirectory
getVersion	contentStore
getWorkItem	global function
hasExpired	workItem
initialize	contentStore
isException	contentStore
isStateActive	workItem
isStateRunning	workItem
isStateSuspended	workItem
logErrorMsg	global function
logHistoryMsg	global function
logInfoMsg	global function
logSecurityMsg	global function
mapTo	global function
mkdir	contentStore
modifyUserByCN	corporateDirectory
modifyUserByDN	corporateDirectory
modifyUserById	corporateDirectory
mount	global function
move	contentStore
moveTo	workItem

JavaScript method or function	Where Used in PAE
randomToGroup	predefined script
resume	workItem
rmdir	contentStore
setConnector	global function
setData	processInstance
setErrorMessage	global function
setErrorMsg	global function
setExpirationDate	workItem
setPriority	processInstance
setRedirectionURL	global function
setRoleByDN	processInstance
setRoleById	processInstance
setTitle	processInstance
store	contentStore
suspend	workItem
toCreator	predefined script
toGroup	predefined script
toManagerOf	predefined script
toManagerOfCreator	predefined script
toManagerOfRole	predefined script
toParallelApproval	predefined script
toUserById	predefined script
toUserFromField	predefined script
upload	contentStore

Migrating from Previous Releases

This appendix describes how to run or edit Process Manager 1.x applications in the PAE 4.0.

This appendix contains the following topics:

- Getting Started
- Migrating SSJS-specific Objects
- Migrating Custom Fields

Getting Started

Before users can begin working with a Process Manager 1.x application, you must first import the application into PAE 4.0's Process Builder, then deploy that application to a cluster. The major steps are as follows:

1. Import the application to Process Builder.
2. Run Check Errors on the application.
3. Add an exception node, if necessary.

4. If the application uses SSJS-specific objects, you must perform the recommended migration action specified in “Migrating SSJS-specific Objects” on page 486.
5. Deploy the application as usual.

Importing an Application to Process Builder

To begin migrating an application to PAE 4.0, you must import the Process Manager 1.x application to Process Builder by performing the following steps:

1. From the Application menu of Process Builder, choose Import from ZIP.

The Import Application from Zip File dialog box appears.

2. Using the Browse button, navigate to the Applications folder where your Process Manager 1.x files are stored.
3. Select the application you want to open.

The application file name should begin with `wf` and end with `.zip`.

4. From the Application Folder drop-down menu, choose the folder where you want Process Builder to extract the application files.
5. Click OK.

Process Builder expands the `.zip` file into the selected folder.

Assigning Exception Nodes

In Process Manager 1.x, exception nodes were required when your application used subprocesses. However, in PAE 4.0, these nodes are assigned to every step in the process. Therefore, an exception node must be assigned to each step in your imported process. When you import a Process Manager 1.x application, Process Builder detects that the application is lacking exception nodes and will automatically create a default exception and assign each step to this exception node for you.

The application is now in PAE 4.0 format and you can open it as usual.

Checking for Errors

Now that your Process Manager 1.x application has been imported to PAE 4.0, you must run Check Errors before you can deploy the application for use. To run Check Error, perform the following steps:

1. Open the application in Process Builder.
2. From the Application menu of Process Builder, choose Check Errors.

The Checking Application dialog box appears.

3. The Messages window appears, displaying any errors, warnings, or information pertaining to the application.

You may see exceptions for JavaScript errors in the Messages window. Previous versions of Process Manager did not perform automatic syntax checking.

If JavaScript errors appear regarding email notification, be sure you have selected both a Content Type (text/html or text/plain) and a Character Set (e.g., us-ascii) in the Inspector window for the notification.

If a “No exception node” exception appears in the Messages window, you must manually assign an exception node to each step in your imported process (see “Assigning Exception Nodes” on page 484). To do so, perform the following steps:

1. Drag an exception node from the Palette to your process map.
2. Double-click or right-click the new node and enter a name for the node.
3. Right-click an action in your process map to bring up the Properties window.
4. From the Exception Manager drop-down menu, choose the new exception node.

The action is now assigned to the exception node you created in Step 1.

5. Repeat Step 3 and Step 4 for each action in your process map that is not assigned to an exception node.

Assign a form to the new exception node. This process is described in Chapter 7, “Designing Forms.”

6. Click Save to save the changes to your application.

Deploying the Application

When you have successfully imported your Process Manager 1.x application, checked for errors, and assigned exception nodes as necessary, and if your application does not use SSJS-specific objects or custom fields written in JavaScript, you are ready to deploy your application as usual. See Chapter 10, “Deploying an Application” for instructions.

If your application *does* use SSJS-specific objects or custom fields written in Java, see the following sections, “Migrating SSJS-specific Objects” and “Migrating Custom Fields” for more information.

Migrating SSJS-specific Objects

Server Side JavaScript (SSJS) is no longer supported by PAE. You can still use standard JavaScript in your applications, but if you imported an application developed with SSJS-specific objects, you must rewrite those portions of your application. The following table describes the recommended migration path from SSJS-specific objects to code compatible with PAE 4.0:

SSJS object	Recommended migration
Database DbPool Connection ResultSet StoredProc Cursor	<p>Each of these objects relates to database access. In PAE 4.0, you should access the database with Java DB access methods (JDBC) using the Custom Activity or Custom Fields features or using JavaScript to access the JDBC methods using Live Connect. For more information on Live Connect, see http://developer.netscape.com/docs/manuals/js/core/jsguide</p> <p>Note that this does not concern access to the main process instance database in the course of normal process development or operation. PAE handles this access transparently. Rather, the SSJS objects listed in the left column were occasionally used in Process Manager 1.x to access databases external to PAE or in other extraordinary circumstances.</p>
File	Use Custom Activities to mimic the File Object functionality. You can also use <code>java.io.file</code> .
SendMail	Use Custom Activities to mimic the SendMail functionality. You can also use the SMTP SDK.
Project Server Request Client	These objects were often used to store data for server-side information. You can now store information in the Process Instance data or in a Custom Field.

More information, see Chapter 17, “Writing Custom Activities.”

Migrating Custom Fields

Developers should note that Custom Fields in PAE 4.0 are different than those in Process Manager 1.x. If your imported application uses Custom Fields, you may have to rewrite portions of code.

In Process Manager 1.x, Custom Fields were written in JavaScript. In PAE 4.0, they are written in Java. There are two parts of a Custom Field: a `.jsb` file, also called a properties file, that contains information about a field, and a `.js` file,

also called an implementation file. The `.jsp` or `properties` file remains the same in PAE 4.0 as in Process Manager 1.x, but you must substitute the content of the `.js` file with a Java implementation.

Reserved Words

This appendix lists reserved words in PAE.

The following words are used internally by PAE and must not be used as the names of data fields. Using reserved words as field names may generate an error when an application is deployed.

Reserved Words in PAE					
abort	cascade	datetime	for	on	start
accept	case	dba	force	open	stop
add	change	dec	form	or	table
admin	char	decimal	format	order	tables
all	character	declare	from	positive	tablespace
allocate	check	default	function	project	task
alter	checkpoint	definition	global	public	temporary
analyse	client	delay	go	quota	text
and	close	delete	goto	raise	then
any	cluster	delta	grant	range	this
append	cobol	disable	group	real	time
archive	column	dismount	groups	rename	trigger
array	columns	dispose	if	replace	true
as	comment	distinct	image	request	type
assert	commit	do	index	resume	union
assign	compile	double	indexed	reverse	unique
at	compress	drop	insert	revoke	unlimited
audit	connect	each	int	role	update
authorization	constant	else	key	roles	use
avg	constraint	enable	layer	row	user
backup	contain	end	link	schema	using
badfile	contents	entry	list	server	values
become	continue	erase	log	set	varchar
before	constraints	escape	longtext	share	varchar2
begin	count	events	manual	shared	variance
between	crash	exit	max	size	view
blob	create	explain	min	smallint	views
block	current	false	new	snapshot	when
body	cursor	file	next	some	while
boolean	cycle	fixed	not	sort	with
by	data	float	number	space	xor
cache	database	flush	off	sql	yes
cancel	date				

Glossary

activity	A step in an application where an assignee needs to perform an action.
application	In Process Builder, the application the builder creates to handle a process.
applet	A Java component designed to run in a web browser.
assignee	The person assigned to an activity for a particular process instance.
automated activity	A step in an application where an action is performed automatically, without being assigned to a user.
bean	A reusable software component for visual development environments.
builder	The person who creates the application using Process Builder.
child process	In subprocesses, the subordinate process that is called by the main or parent process.
class ID	The identifier of a group of fields with some common properties.
CGI	Common Gateway Interface. The specification for communication between an HTTP server and gateway programs on the server. Allows web interfaces to databases and enables the dynamic generation of HTML documents by gateway programs.

cluster	The combination of a configuration directory, a corporate user directory, a relational database, a web server, and one or more Netscape Application Servers. This combination of components is the environment for deployed applications, which are run by the application servers.
configuration directory	The Directory Server where PAE application and cluster information is stored.
content store	The place on the Enterprise Server where file attachments are stored, and the user and password needed to access them.
corporate user directory	The Directory Server used to store the user and group information for a corporation. PAE uses it to set up users and groups and assign users work items by leveraging the directory users and groups and other attributes.
creator	The person who initiates a process instance. Sometimes called the initiator.
custom activity	
database	The database you are using to store the information generated by process instances. For example, the database could be Informix, Oracle, or Sybase.
decision point	A point at which a process map branches depending upon conditions defined in the decision point.
deploy	To copy an application stored locally to a cluster. It can be deployed for storage only, or it can be deployed for development or production. Application information is deployed to the configuration directory, and the application is activated on the Netscape Application Server.
entry point	A point in the process where a user can initiate a process instance.
exception handler	Used in subprocesses, a step in an application that allows the administrator to intervene manually if errors occur in the interaction between a parent and child process.
exit point	A point in the process where the process ends.
extranet	An extension of a company's intranet onto the Internet, to allow customers, suppliers, and remote workers access to the data.
form	A part of an application a user fills out to complete a process instance, or uses to view information on a process.

group	A set of users set up either in the corporate user directory or in an individual application. A group is used to assign work items to users and control which forms which users are able to see.
HTML	HyperText Markup Language. A markup language (derived from SGML) used to create web documents.
HTTP	HyperText Transfer Protocol. A protocol for communication between web clients and servers.
initiator	The person who initiates a process instance. Sometimes called the creator.
intranet	A network which provides similar services within an organization to those provided by the Internet outside it but which is not necessarily connected to the Internet.
LDIF	A format for storing directory entries.
nested parallel process	A parallel process nested within a larger parallel process. The activities in the nested process are considered to be part of the nested process and not the larger process.
PAE	An acronym for Process Automation Edition.
parallel processing	A step in an application that branches between two or more branches so that two or more activities can execute in parallel.
parent process	In subprocesses, the main process that calls the subordinate or child process.
participant	A user of Process Express.
Process Administrator	The component of PAE that administrators use to administer PAE and PAE applications. Process Administrator is the IT administrator's interface described in the <i>Administrator's Guide</i> . Process Business Manager is the business manager's interface, described in the <i>Business Manager's Guide</i> .
Process Automation Edition	The Netscape process management solution.
process	A process is a series of steps, or work items, that can be completed by participants using an application.
Process Builder	The component of PAE where you can design and deploy applications.

Process Business Manager	The component of PAE that business managers use to administer PAE work items and process instances. Process Business Manager is the business manager's interface, described in the <i>Business Manager's Guide</i> . Process Administrator is the IT administrator's interface described in the <i>Administrator's Guide</i> .
Process Engine	The part of PAE that contains Process Express, Process Administrator, Process Business Manager and the engine that runs PAE. It contains all PAE components, with the exception of Process Builder.
Process Express	The component of PAE that end user use to initiate process instances, complete work items, and search for process instances.
process instance	A particular example of a process; for example, in a time off process application, a process instance would be a particular request by an employee for vacation time off for a specific period of time.
process map	The visual representation of the process that is handled by a PAE application.
processing branch	A set of activities that progress from a given split to its corresponding join. Also called a thread.
property	An attribute of an item or component used in an application that contains information about the item. For example, an activity has properties containing information such as the name of the activity, what script is run when it is completed, and so on.
role	A role is the part a user plays in a specific process instance.
script	A JavaScript file. A script can include a function, but this is not a defining characteristic.
subprocess	A fully functional process that is called from within another process. The process that calls the subprocess is the parent process and the subprocess is its child process.
trusted user	A group that allows a secure handshake between a parent and a child process.
URL	Uniform Resource Locator. The address system used by servers and clients to request web documents. It is often called a "location." The format of a URL is:

`[protocol]://[machine:port]/[document]`

For example:

`http://home.netscape.com/index.html`

transition	The links between steps in a process. On the process map, they are represented by lines with arrows that lead from one item to another. Transitions can be regular or they can depend upon a condition being true before they are executed.
web publishing	A feature that lets users access and manipulate server files with a server client, so that they can edit and publish documents to the web server.
work item	An individual task in a process instance as it appears to the end user on a work list.

Index

Symbols

`__getIncludePath()` 471
`__includeFile()` 473
`__setIncludePath()` 476

A

activities
 automated 84
 Data Sheet sample 217
 expiration 79
 planning assignments 54
 setting assignments 81
address field 152
addUser() 444
admin
 forms 174
 group 113
all group 113
applet fields 142
application menu 43
application tree view 38
applications
 creating 63
 deleting 71
 editing a deployed application 207
 folder 65
 getting path 468
 initializing 348
 redeploying 212
 samples 68
 selecting 34
 shutting down 348
archive() method 413
arrays

 example of creating
assignees
 getting 428
assigning
 work items 427
assigning exception nodes 484
assignment scripts 179, 456
 example 337, 352
 predefined 181
assignments
 forms 173
assignTo() 427
attributes
 modifying for users 337
automated activities
 Data Sheet sample 219
 deferring 85
 planning 52
 properties 84
automation scripts 179
 example 355
 example of writing to content store 341

B

banners 171
base file name
 getting for content store 468
BasicCustomField class 413
bean fields 144
buildDataSheet script 231
buildDS script 229

C

- cache
 - allow for application group 118
 - allow for corporate group 122
 - allow for dynamic group 126
- checkbox fields 137
- checkParallelApproval() 456
- checkUserDNs() 338, 464
- class IDs
 - creating your own 151
 - predefined 137
- client-side scripts 191
 - onSubmitForm function 344
 - verifying form input 343
- cn
 - data field property 395
- comments, allow to add 78, 99
- completion scripts 179, 456
 - example 335, 354
 - parallel approval 84
 - predefined 84, 184
 - verifying form input 347
- computed fields 138
- computeTitle script 234
- condition scripts 179
- conditions, for transitions 108
- content store
 - accessing 339
 - example of saving file 341
 - getting base file name 468
 - getting content 436
 - getting root URL 438
 - making directories 441
 - moving files 441
 - setting up 156
 - storing 443
 - storing files 341
- content type, in email notifications 104
- contentStore object
 - reference 433

- copy() 434
- corporate user directory
 - defining for application 67
- corporateDirectory object 444
- create() method 411
- creator 113
- Credit History
 - configuring 297
 - data fields 293
 - description 68
 - exit points 287
 - groups and roles 292
 - scripts 295
 - trusted users 292
- custom activities 366
 - deferring 93
 - planning 52
- custom data fields 134
- custom fields 389
 - migrating 487
 - packaging 405
 - steps for creating new class 391

D

- data fields
 - form elements for 191, 343
 - getting values 335, 419
 - required properties 395
 - setting values 335, 423
- data mapping
 - Loan Management sample 283
- Data Sheet
 - activities 217
 - automated activity 219
 - configuring 241
 - decision point 222
 - description 68
 - description.txt file 235
 - entry point 216
 - exit points 223
 - fields 224
 - finished example 239

- forms 226
- image.gif file 236
- process map 214
- scripts 229
- date fields 138
- dates
 - adding days example 353
- datetime fields 139
- debugging hints
 - for scripts 349
- decision point scripts 179
 - getting data field values 336
- decision points
 - Data Sheet sample 222
 - planning 53
 - properties 100
- defaultNotificationHeader() 185, 457
- defaultNotificationSubject() 185, 457
- deferring automated activities 85
- deferring custom activities 93
- delegation 78, 99
 - planning 57
- deleteUserByCN() 445
- deleteUserByDN() 446
- deleteUserById() 446
- deleting users 445
- deploying 202
 - planning 60
- description
 - data field property 395
- description.txt file 235
- DESIGN tag 378
- destroy() method 369
- development hints
 - for scripts 349
- dialog boxes
 - for script failures 350
- digital signature fields 139
- directories
 - making in content store 441
- Directory Server 445
 - adding users 444
 - getting users 447
 - methods for accessing 444
 - modifying attributes 448
- display() method 409
- distinguished names 338
 - getting process instance creator dn 419
- dn
 - see distinguished names
- download() 434

E

- edit menu 45, 167
- EJB components 360
- ejbLookup() 465
- email notification scripts 105, 179
- email scripts 456
- emailByDN() 185, 458
- emailById() 186, 458
- emailOfAssignees() 186, 459
- emailOfCreator() 186, 459
- emailOfRole() 186, 459
- entry points
 - access to forms 174
 - Data Sheet sample 216
 - Loan Management sample 279
 - Office Setup sample 249
 - planning 51
 - properties 77
- ENVIRONMENT tag 374
- error checking
 - Java Script exceptions 485
- error checking in Process Builder 485
- error messages 342
 - for script failures 350
 - viewing 350
- evaluateTemplate() 465

- event handlers 191
 - example 344
- exception manager 97
 - properties 99
- exception nodes
 - assigning 485
 - Credit History sample 284
- exception nodes, assigning 484
- exists() 435
- exit points
 - Credit History sample 287
 - Data Sheet sample 223
 - Loan Management sample 291
 - Office Setup sample 264
 - planning 53
 - properties 105
- expiration 79
- expiration date
 - setting 432
- expiration handler scripts 179, 351
 - example 354
- expiration setter scripts 179, 351
 - example 353
- expire() 427
- expireIn() 466
- expiring
 - work items 427
- extend() 428
- extending
 - work item expiration date 428

F

- fieldclassid
 - data field property 395
- fields
 - address 152
 - allowing searches for 200
 - checkbox 137
 - class ID 134
 - computed 138

- creating 132
- Credit History sample 293
- custom 134
- Data Sheet sample 224
- data types 136
- date 138
- datetime 139
- deleting 155
- digital signature 139
- display mode 170
- file attachment 140
- help message 136
- Java applet 142
- Java bean 144
- Loan Management sample 292
- name 153
- Office Setup sample 265
- password 144
- planning 57
- predefined 135, 151
- radio buttons 145
- roles 128
- select list 146
- telephone 154
- template 135
- textarea 147
- textfield 148
- URL 149
- userpicker widget 150

- fieldtype
 - data field property 395
- file attachment fields 140
- files
 - moving in content store 441
 - storing in content store 341
- form elements 343
 - for data fields 191
 - verifying input 342
- format menu 45, 168
- forms
 - allowing searches in 198
 - banners 171
 - comments on 165
 - creating 160

- customizing Office Setup samples 273
- Data Sheet sample 226
- display mode 170
- modifying 165
- onSubmitForm function 344
- planning 58
- setting access to 172
- verifying input 342

forms[] array 191

G

- getAction() 467
- getApplicationName() 468
- getApplicationPath() 468
- getApplicationPrettyName() 468
- getAssigneesDN() 428
- getBaseForFileName() 339, 341, 468
- getBaseURL() 435
- getConclusion() 418
- getConnector() 469
- getContent() 436
- getContentStore() 339, 433, 469
- getCorporateDirectory() 336, 444, 471
- getCreationDate() 418, 429
- getCreatorDN() 419
- getCreatorUser() 419
- getCreatorUser() method 337
- getCreatorUserId() 470
- getCurrentActivityCN() 429
- getData() 419
- getData() method 335
- getDN() 451
- getEntityKey() 400
- getEntryNodeName() 420
- getException() 437
- getExitNodeName() 420
- getExpirationDate() 429

- getInstanceId() 421
- getIndiNamingContext() 471
- getName() method 415
- getPrettyName() method 416
- getPriority() 421
- getProcessInstance() 334, 418, 472
- getRoleDN() 421
- getRoleUser() 422
- getRoleUser() method 337
- getRootURL() 438
- getSize() 438
- getStatus() 438
- getSubProcessInstance() 472
- getTitle() 422
- getUserByCN() 447
- getUserByDN() 447
- getUserById() 448
- getUserId() 451
- getVersion() 439
- getWorkItem() 334, 426, 472

global functions

- checkUserDNs() 338
- getBaseForFileName() 339
- getContentStore() 339
- getCorporateDirectory() 336
- getProcessInstance() 334
- getWorkItem() 334
- logErrorMsg() 452
- logHistoryMsg() 453
- logInfoMsg() 453
- logSecurityMsg() 454
- miscellaneous 464
- setErrorMsg() 455

groups 112

- application group 117
- corporate group 121
- creating 114
- defaults 113
- deleting 130
- dynamic group 125

- planning 55
- prioritizing 128

H

- hasExpired() 429
- help
 - data field property 395
 - menu 47
 - messages for fields 136
- helpful messages
 - explaining script failures 350
- hidden form elements
 - for data fields 191
- HTML editor 165

I

- id
 - getting 421
- IDataElement 400, 410
- image.gif file 236
- init() method 367
- initialization scripts 179, 187, 348
- initialize() 439
- initializing
 - applications 348
- INPUT tag 375
- insert menu 45, 167
- instance id
 - getting 421
- IPMElement interface 415
- IPresentationElement 400, 409
- isException() 440
- ISimpleWorkPerformer 367
- isStateActive() 430
- isStateRunning() 430
- isStateSuspended() 431

J

- Java applet fields 142
- Java bean fields 144
- jpeg file
 - saving 203
- JSB File 391
- JSB_DESCRIPTOR tag 392
- JSB_PROPERTY tag 393, 394

L

- LDAP
 - filters 126
- list() 440
- load() method 412
- loadDataElementProperties() method 414
- Loan Management
 - configuring 296
 - data fields 292
 - data mapping 283
 - description 68
 - entry point 279
 - exit points 291
 - groups and roles 292
 - scripts 293
 - subprocess 281
- local application folder 65
- logErrorMsg() 452
- logging error messages 342
- logHistoryMsg() 453
- logInfoMsg() 453
- logs
 - viewing 350
- logSecurityMsg() 454
- lookupCode script 230

M

- mapTo() 474
- menus 43

- messages
 - explaining script failures 350
- migration
 - custom fields 487
 - error checking 485
 - exception nodes 484, 485
 - importing applications to Process Builder 484
 - Java Script exceptions 485
 - server-side Java Script objects 486
- miscellaneous global functions 464
- mkdir() 441
- modifyUserByCN() 448
- modifyUserByCN() method 337
- modifyUserByDN() 449
- modifyUserByDN() method 337
- modifyUserById() 450
- modifyUserById() method 337
- mount() 474
- move() 441
- moveTo() 431
- moving files
 - in content store 441

N

- name fields 153
- notifications 104
 - built-in scripts 105
 - planning 54
- numbers
 - checking for 355

O

- Office Setup
 - configuring 274
 - data fields 265
 - description 68
 - entry point 249
 - exit point 264
 - form customization 273

- forms 267
- groups and roles 264
- parallel processing split 253
- scripts 269
- onClick event handler 343
- onCompletion scripts
 - see completion scripts
- onSubmitForm() 344, 346
- onValueChange event handler 343
- OUTPUT tag 376

P

- palette 39, 75
- parallel approval 82
- parallel processing 101
 - Office Setup example 253
 - planning 53
 - properties 102
 - using 102
- PARAMETER tag 377
- parseInt() 345, 355
- password fields 144
- path
 - getting for application 468
- perform() method 368
- pid
 - getting 421
- predefined data fields 151
 - creating 135
- pre-defined scripts 456
- preferences.ini file 31
- prettyname
 - data field property 395
- priority field 67
- Process Administrator
 - common information 63
- Process Builder
 - starting up 31
- process instances 334

- how they use the content store 341
- id 421
- process map
 - creating 74
 - Credit History sample 277
 - Data Sheet 214
 - deleting items from 77
 - Loan Management sample 276
 - planning 50
 - scripts available 180
- processInstance object 334, 418
- public password 157
- public user 157

R

- radio button fields 145
- randomToGroup() 460
- remove() 442
- required properties
 - of data fields 395
- resume() 431
- rmdir() 442
- roles 112
 - creating 114
 - defaults 113
 - deleting 130
 - field role 127
 - planning 55
 - prioritizing 128
- root URL
 - of content store 438

S

- sample applications
 - configuring
 - Credit History 297
 - Data Sheet 241
 - Loan Management 296
 - Office Setup 274
 - Credit History 276

- Data Sheet 214
- Insurance Claim 300
- list of 68
- Loan Management 276
- Office Setup 248
- save work item 78
- saving a jpeg file 203
- scripts 177
 - accessing current process 334
 - assignment 181
 - client-side 191
 - Credit History sample 295
 - Data Sheet sample 229
 - debugging hints 349
 - displaying progress of 350
 - explaining failures 350
 - getting information about users 336
 - initialization 187, 348
 - Loan Management sample 293
 - Office Setup sample 269
 - setting work item assignees 427
 - shutdown 187, 348
- searches 196
 - allow for application group 118
 - allow for corporate group 122
 - allow for dynamic group 126
 - planning 60
- select list fields 146
- server-side Java Script object
 - migrating 486
- setConnector() 475
- setData() 423
- setData() method 335
- setEntityKey() 400
- setErrorMessage() 350
- setErrorMsg() 455
- setExpirationDate() 432
- setPriority() 424
- setRedirectionURL() 476
- setRoleByDN() 425
- setRoleById() 425

- setTitle() 426
- shutdown scripts 180, 187, 348
- signature fields 139
- SSL
 - enabling 140
- starting Process Builder 31
- store() 443
- store() method 341, 411
- submitting forms 344
- subprocesses 86
 - connecting the parent and child 90
 - Credit History sample 281
 - planning 52
 - properties 88
 - setting up 87
- suspend() 432

T

- telephone fields 154
- templates
 - fields 135
- textarea fields 147
- textfield fields 148
- timer agents 87
- title field 67
- toCreator() 460
- toGroup() 461
- toManagerOf() 461
- toManagerOfCreator() 462
- toManagerOfRole() 462
- toolbar 42
- toolkit scripts 180
- toParallelApproval() 462
- toUserById() 463
- toUserFromField() 464
- transitions 106
 - condition 108
 - planning 54

- troubleshooting
 - scripts 349
- trusted users
 - Credit History sample 292

U

- update() method 410
- upload() 443
- URL fields 149
- url_OnDisplayHistory() 477
- url_OnDisplayProcessInstance() 477
- url_OnDisplayWorkItem() 478
- url_OnDisplayWorklist() 478
- url_OnListApplications() 478
- url_OnListEntryNodes() 478
- user activities
 - properties 78
- user dns
 - verifying array of 338
- userpicker widget fields 150
- users
 - adding to directory server from scripts 444
 - deleting 445
 - getting 447
 - getting information about 336
 - modifying attributes 448
 - modifying attributes 337

V

- verifying form input 342
 - in completion scripts 347

W

- window menu 46
- work items
 - assignTo() 427
 - expiring 427
 - extending 428
 - getting assignees 428

reassigning 427
workItem object 334, 426
WORKPERFORMER tag 374