IBM® Workplace Forms™ Server — Deployment Server

Version 2.6.1

Administration Manual

G325-2594-00

Note

Before using this information and the product it supports, read the information in "Notices," on page 51.

First Edition (September 2006)

This edition applies to version 2.6.1 of IBM Workplace Forms Server — Deployment Server (product number L-DSED-6RFRFZ) and to all subsequent releases and modifications until otherwise indicated in new editions.

This edition replaces version 1, release 2.6.0 of Workplace Forms Server — Deployment Server.

Introduction

Welcome to IBM® Workplace Forms™ Server - Deployment Server. Deployment Server uses standard web browser technology to automatically distribute software to your users. It determines what software is on your user's computers, and installs new software or upgrades existing software depending on your needs.

You can easily incorporate Deployment Server into any web site, either by setting up a central distribution page or by setting up automatic downloads on particular pages throughout your site. In either case, Deployment Server makes it easy to distribute new software or to upgrade existing software through a simple web interface.

About This Manual

This manual provides instructions for installing and configuring Deployment Server. Each section explains a different task in the setup process, and progresses from general information about the architecture of Deployment Server to specific details and instructions.

If you are setting up Deployment Server for the first time, you should begin by reading this introduction and the ″Overview of Deployment Server″ . Then refer to ″Setting Up Deployment Server″ for an outline of the setup process. The setting up section will direct you to the correct pages in the manual for more information about each step.

Who Should Read This Manual

This manual is intended for System Administrators who have experience with both scripting and simple programming. Because Deployment Server does not have an administrative front end, you will need to write a number of scripts to configure the system.

If you do not have scripting or programming experience, this manual will provide you with an overview of the installation process and an explanation of how Deployment Server works; however, you may find the configuration sections more challenging to understand.

Document Conventions

This manual uses the following conventions:

< >	Placeholders are enclosed by the less than and greater than symbols. In general, placeholders identify information you must provide. For example, the following path uses a placeholder to indicate the directory in which you installed ACME WebProgram: `c:\<Installation Directory>\ACME\WebProgram\3.0\`

Key Terms

This manual uses a number of key terms. While these terms are defined in the appropriate sections of the manual, they are also used throughout, and may be confusing when first encountered. For your convenience, these terms are defined here:

Alias Aliases are similar to variables in programming. They allow you to store information by creating a name for that information, and then using that name later to refer to that information. Aliases are used in manifests.

Application
Each piece of software you deploy through Deployment Server is called an application. Deployment Server deploys each application as a collection of one or more packages.

Manifest
A manifest is an instruction set that Deployment Server reads when updating a user's computer. There are three types of manifests, each providing instructions for different aspects of the update process. You can also think of a manifest as a ″script″.

Package
Each package is a set of installation files for either a complete application or a portion of an application. Applications are often divided into several packages to allow more control over which portions of an application are installed. For example, if you deploy a spell checker, you may want the user to choose which dictionaries are installed. In this case, you would deploy each dictionary as a separate package that the user could decide whether to install.

Backwards Compatibility

This version of Deployment Server is fully compatible with all previous versions of Deployment Server. Furthermore, it supports all versions of Workplace Forms and PureEdge-branded Viewer products..

Overview of Deployment Server

Deployment Server allows you to deploy software to your users through standard web browsers, including Microsoft® Internet Explorer and Firefox. Deployment Server automates this process, requiring little or no user interaction, and ensures that the right components are installed for each user.

Deployment Server relies on three key components: the Deployment Server applet, which manages the deployment process, the Deployment Server servlet, which passes files to the applet, and the Deployment Server file system, which stores the instructions and the files the applet uses to install the software. The applet runs in the user's web browser, and communicates with the servlet, which is installed on your server.

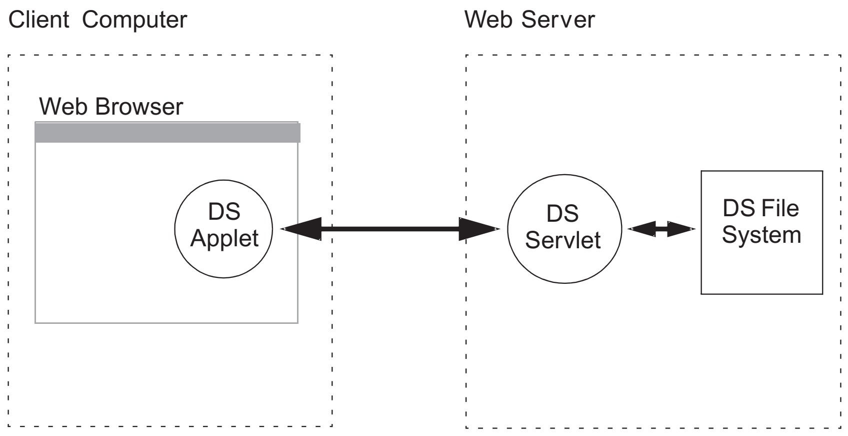
This following sections outline the architecture of the Deployment Server system, and step you through the end-user experience.

System Architecture

Deployment Server has three key components:

- An applet that controls the installation of software on the end-user's computer.
- A servlet that communicates with the applet and passes both instructions and files to the applet.
- A file system that stores all of the instructions and files necessary for deployment.

The following diagram shows how these components work together:



The file system stores two types of files, called *manifests* and *packages*. Each manifest is a set of instructions that the applet follows when deploying an application. Each package is a set of installation files that may install either a complete application or a portion of an application.

A typical software deployment follows these steps:

1. The user opens a web page that contains the Deployment Server applet.
2. The applet runs, and requests updated instructions (the current manifests) from the servlet.
3. The servlet retrieves the manifests from the file system, and passes them to the applet.
4. The applet reads the manifests and checks the configuration of the user's computer.
5. Based on the logic in the manifests, the applet decides which packages to install.
6. The applet requests the necessary packages from the server.
7. The server retrieves the packages from the file system, and passes them to the applet.
8. The applet runs each package in turn.
9. When run, each package installs an application or a portion of an application on the user's computer.
10. The applet monitors the installation, and loads a success or failure page, depending on the results.

During installation, the user will normally see a dialog that lists the components being installed, and may have the option of refusing some or all of the components. Once the installation is complete, the dialog will disappear and the applet will load a success or error page.

End-User Experience

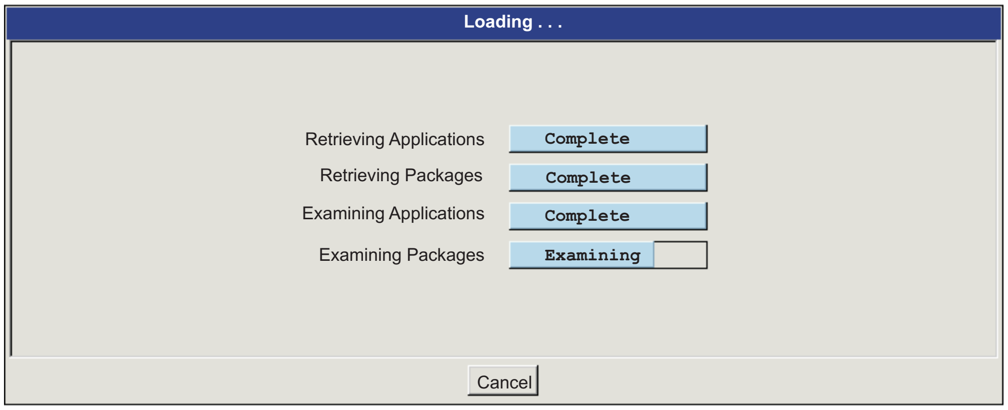
Deployment Server is designed to require minimal interaction from your users. While Deployment Server always shows your users some information about the installation, you can decide whether your users can interact with the installation. This allows you to create completely automated installations.

In general, your users will access Deployment Server through a central distribution site, or as part of an online process (such as opening an account with your organization). Before launching Deployment Server, your web site should detect the user's configuration, and ensure they have Java™ and Javascript enabled.

Your site should then direct the user to a web page containing the Deployment Server applet. If the user has never used Deployment Server, or if you have recently changed or upgraded the Deployment Server applet, they will see a Security Warning asking them if they trust the applet.

The user must accept the applet to continue. If they do not accept the applet, Deployment Server will load a cancellation page in their browser and stop the deployment process.

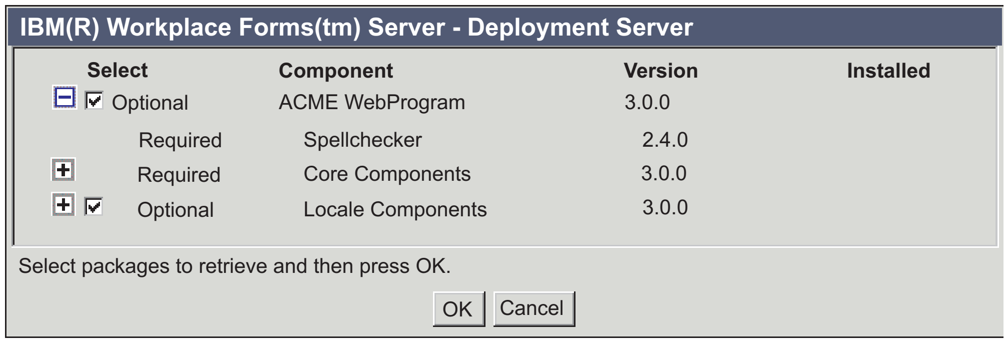
If the user accepts the applet, the browser will load and run the applet, and the applet will initialize itself. While initializing, the applet displays its progress:


Loading . . .

Retrieving Applications	Complete
Retrieving Packages	Complete
Examining Applications	Complete
Examining Packages	Examining

However, initialization does not take long, and in most cases the user will not have time to read the dialog box.

Next, the applet lists the applications and packages that Deployment Server is going to install:


IBM(R) Workplace Forms(tm) Server - Deployment Server

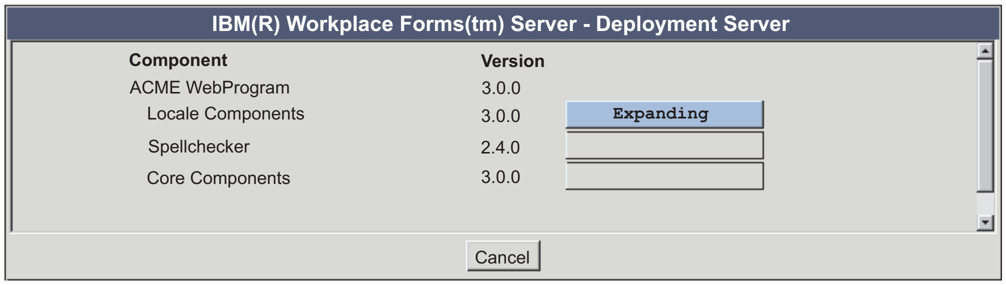
Select	Component	Version	Installed
☑ Optional	ACME WebProgram	3.0.0	
Required	Spellchecker	2.4.0	
Required	Core Components	3.0.0	
☑ Optional	Locale Components	3.0.0	

Select packages to retrieve and then press OK.

At this point, the user may be able to select which packages to install. The applet also shows which version of each application and package Deployment Server is deploying, and which version Deployment Server has detected on the user's computer.

The user can cancel the installation at this stage, or they can select the packages they want and continue.

Deployment Server then begins installing the applications. During installation, Deployment Server displays its progress:


IBM(R) Workplace Forms(tm) Server - Deployment Server

Component	Version	
ACME WebProgram	3.0.0	
Locale Components	3.0.0	Expanding
Spellchecker	2.4.0	
Core Components	3.0.0	

Once Deployment Server has completed the installation, the applet window closes and the applet loads a result page in the user's browser.

Setting Up Deployment Server

Deployment Server runs on a web server using a servlet runner. Before you can set up Deployment Server, you should ensure that you have the right hardware and software configuration on your server. You should also confirm that your users have the necessary configuration on their computers.

Once you have confirmed that you meet the requirements, follow the steps outlined in the setup instructions.

System Requirements

Deployment Server runs both on a server and on your user's computers. Therefore, Deployment Server requires specific software to be installed both on the server and on end-user computers.

For detailed information regarding Deployment Server system requirements, see http://www.ibm.com/support/docview.wss?rs=2357&uid=swg27008291.

Setup Instructions

The setup process varies depending on whether you are deploying applications supplied by IBM, or whether you are deploying your own applications.

For a complete explanation of packages, and the process involved in creating your own packages, see "Using Packages" on page 19.

Setting Up to Deploy Applications Supplied by IBM

If you are deploying applications supplied by IBM, each application will be supplied as a single zip file that contains all of the manifests and packages required to deploy the application. You must copy this file to the server level of your Deployment Server file system, and unzip the file to create the appropriate sub-directories.

The following table outlines the steps for setting up Deployment Server. Each step is explained in detail on the page indicated:

1. "Configuring Deployment Server" on page 9
2. "Using the Deployment Server File System" on page 15
3. Unzip the application zip file (provided by IBM) in the appropriate server directory — this will create the application and package levels of the file system, including the necessary manifest and package files.
4. "Setting Up Your Web Page to Use Deployment Server" on page 43

Setting Up to Deploy Your Own Applications

If you are deploying your own applications, you will need to create a collection of manifests and packages for each application you want to deploy, and then copy those files to the Deployment Server file system. The following table outlines the steps for setting up Deployment Server. Each step is explained in detail on the page indicated:

1. "Configuring Deployment Server" on page 9

2. “Using the Deployment Server File System” on page 15
3. “Creating Your Own Packages” on page 19
4. “Copying Packages to the Deployment Server File System” on page 20
5. “Creating a Master Manifest” on page 23
6. “Creating Manifests” on page 22
7. “Creating a Package Manifest” on page 26
8. “Adding the Manifests to Your File System” on page 42
9. “Setting Up Your Web Page to Use Deployment Server” on page 43

Configuring Deployment Server

The Deployment Server is made up of three major components. These are:

- The virtual server that manages the end-user experience through a number of jsp pages.
- Three applets that manage the interaction between the user computers and the Deployment Server. The user's browser determines which applet is used.
- The servlet that manages the server-side tasks.

Before running Deployment Server, you must configure the system and sign the applets. This allows you to customize Deployment Server to your environment, and authenticates the applet. Java requires authentication of the applet to ensure it has not been tampered with before allowing it to install software onto users' computers. Signing the applet also ensures that it contains the URL of the web server that is providing the deployment service, and informs users who is issuing the software they are accepting.

Note: The Deployment Server is installed as part of the Workplace Forms Server - API installation process. To view the Deployment Server files, open the API installation directory.

The following table outlines the steps you must follow to configure the deployment server and sign the applets:

Step	Section
1. Customize the jsp pages to better reflect your organization.	"Setting Up Your Web Page to Use Deployment Server" on page 43
2. Download the tools that allow you to create and sign .cab files.	"Downloading Additional Tools"
3. Obtain and prepare the code-signing certificates.	"Obtaining Code Signing Certificates" on page 10
4. Update WAR file (WAS 6.0 systems only).	"Updating the WAR file" on page 10
5. Configure the Signing Tool to: • Configure both the servlet and the applets. • Produce two WAR files that are configured and ready for installation on your server. • Produce the applets.	"Configuring the Signing Tool" on page 11
6. Run the Signing Tool.	"Using the Signing Tool" on page 13
7. Install the WAR files produced by the Signing Tool.	"Installing the WAR files" on page 14
8. Copy the applets produced by the Signing Tool to your server.	

Downloading Additional Tools

Before you can use the Signing Tool, you must download and install the following tools:

- **Microsoft Cabinet Software Development Kit —** Allows the Signing Tool to create .cab files.

- **Authenticode for Internet Explorer 5.0 —** Allows the Signing Tool to sign .cab files.

Microsoft Cabinet Software Development Kit

You can download this kit from Microsoft.

Once you have downloaded it, install it on the computer that you will use to sign the applets. You can install it in any directory.

Once installed, locate the cabarc.exe file. You will need to know this location later.

Authenticode for Internet Explorer 5.0

This tool is available as part of the Microsoft .NET Framework SDK. You can download this software from the Microsoft website. Deployment Server supports versions 1.1 and 2.0 of the .NET framework.

Once you have downloaded the SDK, install it on the computer that you will use to sign the applets, then locate the signcode.exe file. You will need to know the location of this file later.

Obtaining Code Signing Certificates

Deployment Server uses two different applets: one for Microsoft browsers and one for Mozilla browsers. Before the applets are run in either browser, the browser will ask your users if they want to trust the applet. Because of this, the applets are signed as part of the configuration process, so that your users know you have approved the use of the applets.

To sign the applets, you will need to obtain a code signing certificate and install it in your local copy of Internet Explorer. You can obtain this certificate from digital certificate vendors, such as VeriSign.

Preparing the Code Signing Certificate

Deployment Server uses the code signing certificate to sign applets intended for use in both IE and Firefox. To sign the IE applet, Deployment Server uses the .spc and .pvk files that are provided by your certificate vendor. However, to sign the Mozilla applet, you must also import that certificate into your Microsft browser and then export it as a PKCS#12 (.pfx) file.

Write down the location of the .pfx file, as you will need to know it later.

Updating the WAR file

If you are using Deployment Server with WebSphere® Application Server (WAS) 6.0, you must update the default Deployment Server WAR file provided with the installation. If you are using another application server or servlet runner, such as WAS 5.1, JRun, or Tomcat, you will not need to perform this task.

To update the WAR file:

1. Open the Deployment Server installation directory.
2. Rename the IDS.WAR file.
   - For example: **IDS-old.war**.
3. Rename the IDS-WAS60.WAR file to IDS.war.

Configuring the Signing Tool

The Signing Tool reads configuration information from a configuration file. The configuration file is a plain text file that contains a series of tag value pairs.

A sample configuration file is provided with Deployment Server. We recommend that you edit this file to reflect your own configuration rather than creating a new file. You can also rename the file as you like if you need to keep separate versions (for example, you will need a different configuration for each virtual server you are running).

The following table lists the tag value pairs used in the configuration file:

Tag	**Value**
applet-install_config. properties.server_url	The URL of the Deployment Server servlet. This URL is hardcoded into the applet, and used to contact the servlet. The Deployment Server servlet is part of the ServerIDS.WAR file that is produced by the Signing Tool. The URL you need to use will depend on the configuration of your servlet runner, but in most cases will be: `http://<server name>/ServerIDS/servlet/ServerIDS`
server-config. properties.rootDirectory	The path to the Deployment Server file system on your server. Deployment Server retrieves all manifests and packages from this location. Use the slash character (/) as a file separator, regardless of your platform. You will have to set up the file system on your server. For more information about the file system, see "Using the Deployment Server File System" on page 15.
microsoft.cabarc.dir	The path to the directory that contains the cabarc.exe file.
microsoft.signcode.dir	The path to the directory that contains the signcode.exe file.
microsoft.dir	The path to the directory that contains your code signing certificate for Microsoft browsers. Use this tag if the certificate is saved to your file system. If the certificate is in the CryptoAPI store, use the microsoft.common.name tag instead.
microsoft.spc	The name of the .spc file for your code signing certificate for Microsoft browsers. Use this tag if the certificate is saved to your file system. If the certificate is in the CryptoAPI store, use the microsoft.common.name tag instead.
microsoft.pvk	The name of the .pvk file for your code signing certificate for Microsoft browsers. Use this tag if the certificate is saved to your file system. If the certificate is in the CryptoAPI store, use the microsoft.common.name tag instead.
microsoft.common.name	The common name of your code signing certificate for Microsoft browsers. Use this tag if the certificate is stored in the CryptoAPI store. Otherwise, use the microsoft.dir, microsoft.spc, and microsoft.pvk tags.

Tag	Value
java.jarsigner.dir	Optional. The path to the directory that contains the jarsigner.exe file. This is installed with the JDK, and is normally in the following location: `c:\j2sdk<version>\bin` If this value is not provided, the Signing Tool tries to locate the jarsigner.exe file on the current PATH.
java.keystore.file	The full path to the file containing a PKCS#12 certificate (exported from either IE or Mozilla). For example: `c:\testcert.pfx`
java.keystore.password	The password for the certificate.

java.cert.name	The name of the certificate. This can be determined by running the keytool program (which is installed with the JDK). Use the following command to run the tool:
	`keytool -list -storetype pkcs12 -keystore`
	`   <path to certificate>`
	This generates a stream of output. The last two lines will look something like this:
	`<name>, 20-Jul0-2005, keyEntry,`
	`   Certificate fingerprint...`
	Note that you must copy the complete name, including any symbols such as parentheses that might appear around it.

info.url	A URL. When users are asked whether they want to trust the applet, they are provided with a link to further information about the company that signed the applet. This URL sets the destination of that link.
server-war.jars.include	A *true* or *false* value, indicating whether you want to use Log4J to log the Deployment Server server activity. If your servlet runner has problems with logging, set this value to *false*. Log4J is an open source toolkit for log creation and management. Log4J is shipped with Deployment Server, and can be customized to your needs. For more information about Log4J and customizing Log4J, go to the Apache web site at: http://jakarta.apache.org/log4j
server-war.jars.name	The path to the Log4J jar file. This is not necessary if you are not using Log4J (that is, if *server-war.jars.include* is *false*).
server-log4j.properties. name	The path to the Log4J properties file. A default properties file is shipped with Deployment Server, or you can use your own. If you use your own properties file, do not include the next four entries in your configuration file.
server-log4j.properties. path1.path	The path and filename Deployment Server will use for the log file. Do not include this if you are using your own Log4J properties file.

Tag	Value
server-log4j.properties. path2.path	The path and filename Deployment Server will use for the debug file. Do not include this if you are using your own Log4J properties file.
server-log4j.properties. path1.name	Must be: log4j.appender.A1.file Do not include this if you are using your own Log4J properties file.
server-log4j.properties. path2.name	Must be: log4j.appender.A2.file Do not include this if you are using your own Log4J properties file.
timestamp.active	A *true* or *false* value indicating whether the signature should be timestamped. Timestamping the signature ensures that the signature remains valid even if the certificate expires.
timestamp.url	The URL of the timestamp server to use when signing the servlet for Microsoft browsers. For example: `http://timestamp.verisign.com/scripts/timstamp.dll` Note that you cannot change the timestamp server used when signing the servlet for Mozilla browsers.

Configuring Applets for Virtual Servers

The Signing Tool embeds the location of the servlet into the applet. This means that you will need to configure each virtual server separately. Furthermore, if you want to use the template web site provided, you should make any changes to the site before using the Signing Tool (for more information, see “About the Template Web Site” on page 45).

Using the Signing Tool

The Signing Tool is a command line tool that configures the Deployment Server servlet and applet, and signs the Deployment Server applet so that your users know you have approved its use.

While the Deployment Server can be installed and configured on any supported operating system, you should sign the applets on a computer on a Windows® platform that has Java version 1.4 available. Once signed, you can copy the files to any of the other platforms supported. This allows you to support users with the widest range of operating systems. However, if you know that you will not be using Deployment Server to install software on Microsoft computers, then you can sign the applet on any computer.

Before running the tool, you must first copy the following files to the same directory on your computer:

- ServerIDS.war
- IDS.war
- IDS-Signtool.jar
- The modified Signing Tool configuration file.

These files are included in the Deployment Server installation package.

When run, the Signing Tool creates two WAR files that contain the properly configured Deployment Server application, two CAB applets for use with Microsoft browsers, and a JAR applet for use with Mozilla browsers.

Running the Signing Tool

To run the Signing Tool, use the following command:

```
java -jar IDS-SignTool.jar <Signing Tool configuration file>
```

The Signing Tool creates a directory called *configured*, which contains the following files:

- ServerIDS.war
- IDS.war
- IDS-IE.cab (applet for Microsoft browsers)
- IDS-NS.jar (applet for Mozilla browsers)

These WAR files and applets are now properly configured and signed, and are ready for installation.

Installing the WAR files

Once you have configured the WAR files, you can install them in your servlet runner. The ServerIDS.war contains the servlet, and the IDS.war contains the template web site and the applets.

If you want to use the template web site, you must install both the ServerIDS.war and IDS.WAR files in the appropriate directory in your servlet runner.

If you want to use your own web site, you must install the ServerIDS.war and the individual applet files in the appropriate directory in your servlet runner. For more information, see “Copying the Applet Files” on page 45.

Refer to the documentation for your servlet runner to determine how to install and activate the WAR files.

If you want to modify the template web site, see “Modifying the Template Web Site” on page 48.

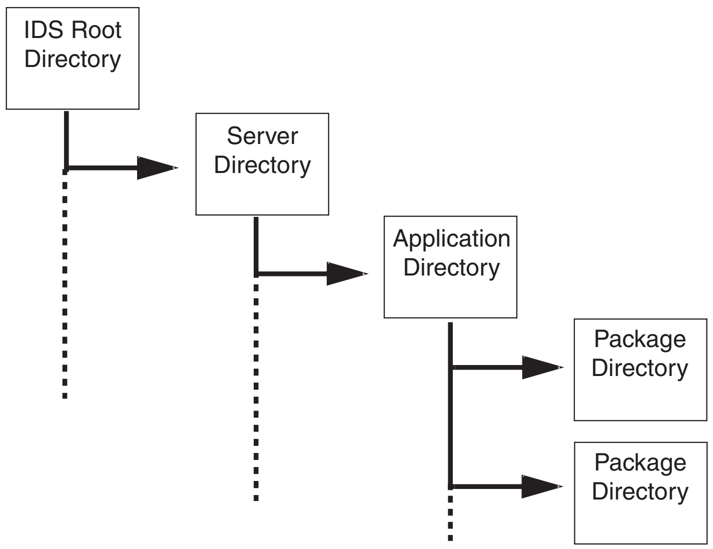
Using the Deployment Server File System

Deployment Server relies on a file system to organize the manifests and packages it uses during the deployment process. This file system provides the structure that Deployment Server uses to determine which applications to check, which manifests to read, and which packages to deploy.

Before using Deployment Server, you must create a file system and copy a complete set of manifests and packages to that system.

About the Deployment Server File System

Deployment Server uses a four level file system to store the manifests and packages required for deploying each application. The file system uses the following directory structure:



Each level in the file system serves a specific purpose and contains specific information:

- **Root Directory** — The root directory is where the Deployment Server servlet looks to find all of the manifests and application packages it will use during deployment. The root directory contains one or more server directories.
- **Server Directories —** Each server directory corresponds to a virtual server running Deployment Server, and contains the master manifest for that server. Each server directory also contains one or more application directories.
- **Application Directories** — Each application directory corresponds to a single application you are deploying, and contains an application manifest. Each application directory also contains one or more package directories.
- **Package Directories** — Each package directory corresponds to a single package you are deploying, and contains a package and a package manifest.

For detailed instructions on setting up the file system, see ″Setting Up the File System″.

How Deployment Server Uses the File System

Deployment Server uses the file system to organize the applications and packages that will be installed by each server. When a user logs on to a server, Deployment Server processes all of the applications and packages in the corresponding server directory in the following manner:

1. Deployment Server reads the master manifest, which contains a list of applications to process.
2. Deployment Server checks the file structure for applications that are not in the list. If it locates additional applications, it adds them to the list in an undetermined order.
3. Deployment Server checks each application manifest for a list of packages to process.
4. Deployment Server checks the file structure for packages that are not in the list. If it locates additional packages, it adds them to the list in an undetermined order.
5. Deployment Server processes each application manifest in the order listed, and determines whether to process the packages for each application.
6. Deployment Server processes each package manifest in the order listed, and determines whether to install the packages.
7. Deployment Server installs all of the necessary packages.

In this way, Deployment Server uses the file system to create the final list of application and package manifests to be processed. However, you may want to list all of the applications and packages in the manifests, since this allows you to specify the order in which they are processed.

Note: Deployment Server will process the manifests for all applications and packages in the server directory. If you do not want an application installed, you will either have to change the period during which the application is active (using the Active tag in the manifest) or remove the application from the server directory.

Setting Up the File System

Before using Deployment Server, you must set up the correct file system on your server, and make sure you have copied the necessary manifests and packages to the file system.

To set up the file system, you will have to create a directory structure that contains the following levels:

- Deployment Server Root Directory
- Server Directories
- Application Directories
- Package Directories

Deployment Server Root Directory

You can create the Deployment Server root directory anywhere on your server, and give it any name. The location of this directory is coded into the Deployment Server servlet during configuration.

The Deployment Server root directory contains one or more server directories.

Server Directories

Each installation of Deployment Server should include a server directory named *default*. The default server directory is used as the general case, and is accessed whenever a specific server directory cannot be located.

You can also create any number of specific server directories. These are useful if you need different servers to install different applications. Each directory name must follow these conventions:

- The directory name must match the server URL used when configuring the applet (that is, the value of the *applet-install_config.properties.server_url* tag). For example, www.IBM.com.
- Replace all periods in the domain name with underscore symbols. For example, ″www.IBM.com″ becomes ″www_IBM_com″.
- Append the correct port number to the domain name, preceded by an underscore. For example, ″www_IBM_com_80″.

Each server directory may also contain the master manifest for that server, and contains one or more application directories.

For more information about creating a master manifest, see ″Using Date Ranges in Manifests″ .

Application Directories

Each application directory represents a single application that may be deployed to the user. The directories can have any name, but you must restrict directory names to alpha-numeric characters (that is, do not use punctuation such as periods or semi-colons).

Each application directory contains the application manifest for that application, as well as one or more package directories.

For more information about creating application manifests, see ″Creating an Application Manifest″.

Package Directories

Each package directory represents a single package that may be deployed to the user. The directories can have any name, but directory names should be restricted to alpha-numeric characters (that is, do not use punctuation such as periods or semi-colons).

Each package directory contains a package manifest and the corresponding package.

For more information about:

1. Creating package manifests, see ″Creating a Package Manifest″.
2. Packages, see ″Using Packages″.

Using Packages

Deployment Server deploys software in the form of *packages*. Each package is zip file that contains:

- The files necessary to install an application or a portion of an application.
- An installation program that will install the files.

In general, Deployment Server will deploy an application as a number of packages. Deployment Server can make decisions about whether to install individual packages, even if those packages are all part of the same application. This allows greater flexibility in determining which components are installed for each user.

For example, you might divide a spell checker into a number of packages: one for the core spell checking engine, and one for each dictionary. When deploying the spell checker, you might set the core engine to be mandatory, but each dictionary to be optional, allowing the user to choose which languages to install.

You can use packages provided by IBM, or you can create your own packages.

Using Packages Provided by IBM

IBM provides complete sets of packages and manifests to deploy products such as IBM Workplace Forms Viewer. Each product or application is delivered as a single zip file which contains:

- All of the manifests required to deploy the application (text files).
- All of the packages required to deploy the application (zip files).

To setup Deployment Server to deploy an application provided by IBM:

1. Copy the application zip file to the appropriate server directory in your Deployment Server file system.
2. Unzip the application file.
   - The application and package sub-directories are created in your file system, and the package and manifest files are automatically copied to the appropriate directories.

Deployment Server will now deploy the application to any user logging on to that server.

For more information about the Deployment Server file system, see ″Using the Deployment Server File System″.

Creating Your Own Packages

If you want to deploy your own software through Deployment Server, you must create your own packages. Each package should contain either a complete application, or a discrete component of an application. For example, you might split a spell checker into one package for the core spell-checking engine and one package for each dictionary file. This would allow you or your users to make decisions about which dictionaries to install.

Once you have determined how to split your application into packages, follow these steps to create each package:

1. Assemble the files you need to install the component.
2. Create an installation package using your preferred software (such as InstallShield).
3. Ensure the installation program is named either *setup.exe* or *install.exe*.
4. Zip the install program and any required files.

The zip file is your Deployment Server package. You can now copy the zip file to the Deployment Server file system.

Copying Packages to the Deployment Server File System

Once you have created your packages, you can simply copy them to the appropriate package directories in your Deployment Server file system. Remember that you must also create a package manifest to match each package.

For more information about the Deployment Server file system, see "Using the Deployment Server File System" on page 15.

For more information about creating manifests, see "Using Manifests" on page 21.

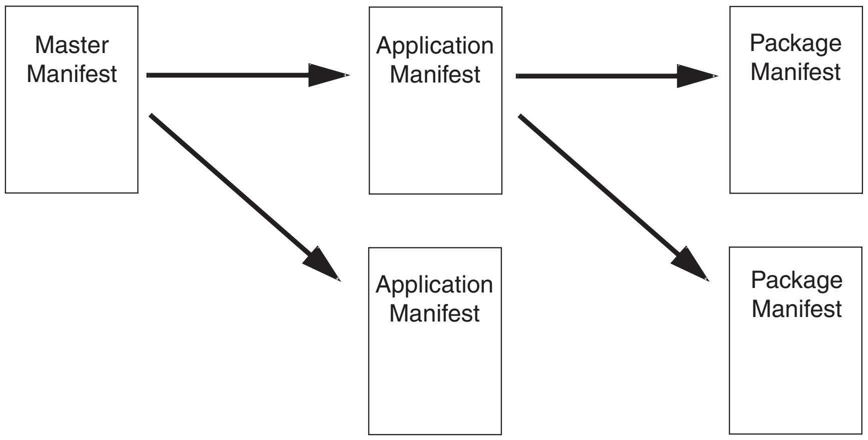
Using Manifests

Manifests provide the instructions that the applet follows when updating the user's computer. These instructions provide the information the applet needs to decide which applications and packages should be updated, and in what order.

Before you create your manifests, you should take some time to understand the manifest hierarchy and how Deployment Server uses manifests to make decisions.

About the Manifest Hierarchy

Manifests rely on a three level hierarchy that includes a single master manifest, one or more application manifests, and one or more package manifests, as shown:



The master manifest controls the order in which the applications are installed, as well as the text that appears in the title bar of the Deployment Server installation window.

The application manifests determine whether an application should be on the user's computer and outline the rules for detecting existing applications. For example, you might need to determine whether a newer version of the application is installed.

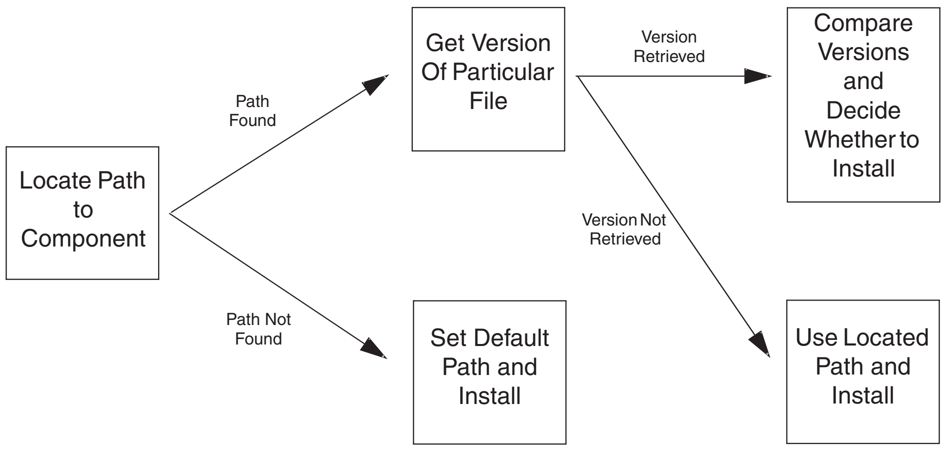
The package manifests determine whether each package will be installed, and control the detection of existing packages in the same way application manifests can detect existing applications.

How Manifests Make Decisions

Both application and package manifests allow you to detect software components on the user's computer and make decisions based on that knowledge.

For example, you might want to upgrade your users to a new version of ACME WebProgram. If they have an old version installed, you'll want to install the new version in the same location. If they don't have the WebProgram installed, you'll want to install the WebProgram in a default location.

Using the decision structure in manifests allows you to perform different actions like these, based on the user's configuration. Both application and package manifests offer the same logical structure, as shown:


Locate Path to Component
Path Found
Path Not Found
Get Version Of Particular File
Set Default Path and Install
Version Retrieved
Version Not Retrieved
Compare Versions and Decide Whether to Install
Use Located Path and Install

First, the Deployment Server attempts to locate the path to a particular component. This is generally retrieved from a registry key.

If the path is found, Deployment Server looks for a specific file in that directory. The name of this file is defined in the manifest.

If the file is found, Deployment Server gets the version number of that file. Deployment Server then compares the version of the component it is deploying to the version of the located file, and determines whether an update is required. If the installed component is out of date, Deployment Server will install the updated component in the located directory.

If Deployment Server is unable to locate a path, Deployment Server sets a default path and installs the component.

If Deployment Server is unable to retrieve a version number, Deployment Server will install the component in the located directory.

This flow is controlled by the *DetermineUpdate* function, which is defined in each application and package manifest.

Creating Manifests

Each manifest is a plain text file containing a series of tag value pairs. Using these tag value pairs, you can define specific values, such as the installation path, or you can create expressions that Deployment Server will evaluate at run time. For example, you might want Deployment Server to detect whether an application is already installed on the user's computer.

All manifests, regardless of type, are saved with the filename *manifest.properties*, and are distinguished from each other by their location in the file system. Each type of manifest has a default set of tag value pairs, some of which are mandatory. Furthermore, manifests may also contain a number of tag value pairs that represent the decisions being made, as well as locale information.

This section describes the default tag values pairs for each type of manifest, and explains how to create expressions that will make decisions.

Using Date Ranges in Manifests

A number of the tags in manifests require a date range as the value. This range must be expressed as <*start date*> - <*end date*>, and each must be a valid date in the Gregorian calendar. Other calendar formats, such as Chinese or Hebrew, are not supported by the Deployment Server.

Furthermore, the date must be in the following format:

```
YYYY/MM/DD HH:MM:SS
```

For example:

```
2002/01/01 00:00:00 - 2002/12/31 23:59:59
```

would represent the period of 12 am, Jan. 1, 2002 to 12 pm, Dec. 31, 2002.

You can also replace the dates with the key words *start* and *end* as follows:

- If you replace the start date with the word *start*, the range will include all time before the end date.
- If you replace the end date with the word *end*, the range will include all time after the start date.
- If the value is *start - end*, the range includes all time (that is, from the beginning of time to the end of time).

Creating a Master Manifest

The master manifest sets the order in which applications are installed and the title that appears in the status window while the Deployment Server applet is running.

A master manifest contains the following tag value pairs:

Tag	**Value**
Version (mandatory)	This is the version of the manifest scripting language used to write the manifest. The current version is *1.0*.
NameList (optional)	A comma separated list of applications. In this list, the name of any application must be the same as the folder name for that application in the Deployment Server file system. The applications will be installed in the order listed. Any applications in the file system that are not included in the list will be installed after the listed applications, in an undetermined order. If no *NameList* is available, all applications in the file system will be installed in an undetermined order.
Title (optional)	A string that is displayed at the top of the status window while the applet is running. Your users will see this status window while the applet is updating their computer. The default value is ″IBM Workplace Forms Server - Deployment Server″.

For example, a master manifest might look like this:

```
Version = 2.0
NameList = Acme_WebProgram, Acme_DesktopProgram
Title = Acme Staff Deployment
```

If no master manifest is available, the applications will be installed in an undetermined order, and the applet will use the default title of *IBM Workplace Forms Server - Deployment Server*.

Creating an Application Manifest

The application manifest determines whether Deployment Server should process the package manifests for that application. In making this decision, it follows these rules:

- If the existing application is either a lower version or the same version, then the package manifests should be processed. This ensures that minor updates to specific packages are deployed, even if the overall application version has not changed.
- If the existing application is a higher version, then the packages manifests should not be processed.

For example, if the user has version 2.0 of an application, and Deployment Server is deploying version 3.0, then the 3.0 packages should be processed. Likewise, if the user already has version 3.0, the 3.0 packages should still be processed in case some of them contain minor updates.

The application manifest also determines the order in which packages are installed. This may be important if some packages, such as core components, must be installed before other packages.

An application manifest contains the following tag value pairs:

Tag	Value
Version (mandatory)	This is the version of the scripting language used to write the manifest. The current version is *1.0*.
ApplicationName (mandatory)	This is the name of the application that Deployment Server will display to the user in the installation dialog. This string can include spaces and punctuation, and can be of any length. However, keep in mind that if application names are too long, the user will have to use a horizontal scroll bar to read them.
ApplicationVersion (mandatory)	This is the version number of the application Deployment Server is deploying. You must use a four digit version number. For example, 4.0.1.2.
NameList (optional)	A comma separated list of packages. In this list, the name of any package must be the same as the folder name for that package in the Deployment Server file system. The packages will be deployed in the order listed. Any packages in the file system that are not included in the list will be deployed after the listed packages, in an undetermined order. If no *NameList* is available, all packages in the file system will be deployed in an undetermined order.

Tag	Value
Active (optional)	This is a date range that defines the period of time during which Deployment Server will deploy this application. Deployment Server will deploy the application to any user who connects during this time. The default value is *start - end*.
InstallMode (optional)	This tag defines the manner in which Deployment Server will install the application, as well as the time period during which that mode will apply. For example, you might want an application to be optional for the first three weeks, and then required after that (allowing your users some time to upgrade). There are 3 modes available: • **Optional** — Deployment Server will install the application by default. The application is listed in the installation dialog, and the user may choose not to install it. • **Required** — Deployment Server will install the application by default. The application is listed in the installation dialog, but the user may not refuse it. If the installation occurs outside of the specified date range, the application is treated as optional. • **Silent** — Deployment Server will install the application by default. The application is not listed in the installation dialog, and the user may not refuse it. If the installation occurs outside of the specified date range, the application is treated as optional. Note that regardless of the install mode, users will be able to cancel any installation by clicking the *Cancel* button in the installation window. If all applications and packages are silent, Deployment Server will not display the installation dialog that lists the applications and packages. However, the user can still cancel the installation from the progress dialog box, which is always shown. If you define multiple install modes with an overlapping date range, the order of precedence is: • Optional overrules both required and silent. • Required overrules silent. The InstallMode tag is written as *InstallMode.<mode>*. For example, *InstallMode.optional*. The value assigned to the tag is the date range during which that mode is valid. For example: `InstallMode.optional = 2002/01/01 00:00:01 - end` If you do not set an install mode for an application, the install mode is determined as follows: • If all of the packages have the same install mode, the application inherits that mode. • If the packages have mixed install modes, but at least one is *silent* or *required*, the application is *required*. • If the packages do not have an install mode, the application is *optional*.

Tag	Value
Command (mandatory)	The name of the expression that will decide whether to update the application. You can use any name. For example: `Command = MakeDecision` You will also have to define the expression in the configuration file. For more information, see "Adding Expressions to a Manifest" on page 30.

For example, an application manifest might look like this:

```
Version = 1.0
ApplicationName = Acme_WebProgram
ApplicationVersion = 3.0.0
NameList = CoreComponents, Spellchecker, LocaleComponents
Active = start - end
InstallMode.required = start - end
Command = MakeDecision
<MakeDecision expression>
```

For more information about defining the expression, see "Adding Expressions to a Manifest" on page 30.

Localizing the Text Displayed by the Applet

Users can select which language the install pages and applet use by selecting the language they want from the **Language** list in the upper right-hand corner of the install page. However, to ensure that the install applet displays the correct localization properties, you must localize your manifest tags.

To localize your manifest tags:

1. Append the end the manifest tag with the language code for the locale you want to use. For example:
   ```
   ApplicationName.fr
   ```
2. Create a new manifest tag for each language you want to support. For example:
   ```
   ApplicationName.fr
   ApplicationName.de
   ApplicationName.ko
   ```

Note: Only the*ApplicationName* tag and *Commands* that use the *Path* Function can be localized.

Creating a Package Manifest

The package manifest determines whether a package will be installed. For example, if you were deploying version 5.0.1 of a particular component, and the user had version 5.0.0, you would want to install the new component.

If Deployment Server determines that a package should be installed, it will copy that package to the user's computer and run the package's installation program. If Deployment Server determines that a package should not be installed, it will move on to the next package.

A package manifest contains the following tag value pairs:

Tag	**Value**
Version (mandatory)	This is the version of the scripting language used to write the manifest. The current version is *1.0*.
PackageName (mandatory)	This is the name of the package that Deployment Server will display to the user in the installation dialog. This string can include spaces and punctuation, and can be of any length. However, keep in mind that if the package names are too long, the user will have to use a horizontal scroll bar to read them.
PackageVersion (mandatory)	This is the version number of the package Deployment Server is deploying. You must use a four digit version number. For example, 4.0.1.2.
Active (optional)	This is a date range that defines the period of time during which Deployment Server will deploy this package. Deployment Server will deploy the package to any user who connect during this time. The default value is *start - end*.

Tag	Value
InstallMode (optional)	This tag defines the manner in which Deployment Server will install the package, as well as the time period during which that mode will apply. For example, you might want a package to be optional for the first three weeks, and then required after that (allowing your users some time to upgrade). There are 3 modes available: • **Optional** — Deployment Server will install the package by default. The package is listed in the installation dialog, and the user may choose not to install it. • **Required** — Deployment Server will install the package by default. The package is listed in the installation dialog, but the user may not refuse it. If the installation occurs outside of the specified date range, the package is treated as optional. • **Silent** — Deployment Server will install the package by default. The package is not listed in the installation dialog, and the user may not refuse it. If the installation occurs outside of the specified date range, the package is treated as optional. Note that regardless of the install mode, users will be able to cancel any installation by clicking the *Cancel* button in the installation window. If all applications and packages are silent, Deployment Server will not display the installation dialog that lists the applications and packages. However, the user can still cancel the installation from the progress dialog box, which is always shown. If you define multiple install modes with an overlapping date range, the order of precedence is: • Optional overrules both required and silent. • Required overrules silent. The InstallMode tag is written as *InstallMode.<mode>*. For example, *InstallMode.optional*. The value assigned to the tag is the date range during which that mode is valid. For example: `InstallMode.optional = 2002/01/01 00:00:01 - end` Note that packages can have a different install mode than the applications they belong to, and the application's install mode may affect the package. For example, if the application is silent, the application will not be listed in the installation dialog box, and the user will not be able to deselect optional packages. If a package's install mode is not set, it will inherit the install mode of the application it belongs to. If the application also does not have an install mode set, the application's install mode will be set by the default rules, then the package will inherit the application's install mode.

Tag	Value
Command (mandatory)	The name of the expression that will decide whether to install the package. You can use any name. For example: `Command = MakeDecision` You will also have to define the expression in the configuration file. For more information, see "Adding Expressions to a Manifest" on page 30.
Install.list (required)	The filename of the package that Deployment Server will download to the user's computer. For example: `Install.list = spellchecker.zip` The package is retrieved from the appropriate package directory in the file system, which also contains the package manifest.
Install.command (optional)	A string containing the command line parameters that Deployment Server will use when running the installer contained in the package. This is useful if you want to use specific installation features for a particular package. For example: `Install.command = -silent`

For example, a package manifest might look like this:

```
Version = 1.0
PackageName = CoreComponents
PackageVersion = 3.0.0
Active = start - end
InstallMode.required = start - end
Command = MakeDecision
<MakeDecision expression>
Install.list = coreComponents.zip
Install.command = -silent
```

For more information about defining the expression, see "Adding Expressions to a Manifest" on page 30.

Localizing the Text Displayed by the Applet

Users can select which language the install pages and applet use by selecting the language they want from the **Language** list in the upper right-hand corner of the install page. However, to ensure that the install applet displays the correct localization properties, you must localize your PackageName tags.

To localize your PackageName tags:

1. Append the end of each PackageName tag with the language code for the locale you want to use. For example:
   ```
   PackageName.fr
   ```
2. Create a new manifest tag for each language you want to support. For example:
   ```
   PackageName.fr
   PackageName.de
   PackageName.ko
   ```

Note: Only the*ApplicationName* tag and *Commands* that use the *Path* Function can be localized.

About Functions

You use functions to make decisions in a manifest. In other words, the functions provide the logic that Deployment Server uses to determine whether to update the software on the user's computer.

Manifest functions are just like function calls in programming. You call a particular function, and pass it some values. The function then does some work, and returns a value. However, unlike programming, you must call each function using a series of tag-value pairs.

For example, in Java, you might define the following function:

```
Vector DetermineWhetherToUpdate(Vector parameter1, Vector parameter2)
```

This function takes two parameters, both arrays of strings, and returns an array of strings. The function itself would decide whether the software on the user's computer should be updated. The return value might be *true* or *false*, depending on the decision the function makes.

To call the function, you would use the following expression:

```
return = DetermineWhetherToUpdate(stringList1, stringList2);
```

Manifests recreate this same function call as a set of tag-value pairs. The first tag-value pair defines which function you are calling, while the following tag-value pairs define the values you are passing to the function.

For example, you would call the same *DetermineWhetherToUpdate* function using the following expression:

```
return = DetermineWhetherToUpdate
parameter1 = string1
parameter2 = string2
```

The full syntax of the tag-values pairs is actually somewhat more complex (and is more fully described in the next section), but the basic structure directly parallels a function call in C.

The available functions are outlined in ″Function Details″.

Adding Expressions to a Manifest

Each manifest includes a *Command* tag. The value of this tag is the name of the first expression you want to evaluate. For example:

```
Command = <expression name>
```

You can choose any name for an expression, just as you can choose any name for a variable in programming. Expressive names are best, as they make your manifests easier to understand. For example, to call a function that would decide whether to install an application, you might use the following name:

```
Command = MakeDecision
```

Next, you must define the expression. Each expression is written as a series of tag value pairs. When you create an expression, you must indicate what function the expression should call and list the parameters for the function.

For example, a typical expression is written as follows:

```
<expression name> = <function name>
<parameter1> = <value>
<parameter2> = <value>
```

The expression name must always be followed by a ″.function″ extension. This identifies the first tag the beginning of the expression. For example:

```
MakeDecision.function = <function name>
```

The function name is the name of the function you want to call. For example:

```
MakeDecision.function = DetermineUpdate
```

The number of parameters depends on the function you are using. Each parameter begins with the name of the expression, and is followed by the parameter name and the parameter type. For example:

```
MakeDecision.<parameter name>.<parameter type> = <value>
```

The parameter names are defined by the function you are using. For example, the *DetermineUpdate* function has four parameters: *dir*, *file*, *defaultDir*, and *version*. The first parameter is written as follows:

```
MakeDecision.dir.<parameter type> = <value>
```

The parameter type is similar to a data type in programming, and sets the data type for the parameter. Each parameter can contain an array of that data type. For example, if you used the *literal* type, the parameter could contain and array of literal strings, as shown:

```
MakeDecision.defaultDir.literal = <value>
```

For more information about:

1. Parameter types, see ″Parameter Types″.
2. Functions available, see ″Function Details″.

Using Nested Expressions

You can call one expression from another, much like you call subroutines in programming. This allows you to calculate the value of any parameter using another expression. This is useful for creating sophisticated manifests that perform more calculations than the *DetermineUpdate* function can do alone.

To call another expression, you first set the parameter to an expression type by setting the parameter type to *expr*:

```
MakeDecision.defaultDir.expr = <value>
```

You then set the parameter's value to the name of the expression you want to call:

```
MakeDecision.defaultDir.expr = SetDefaultDir
```

Finally, you define the new expression later in the manifest. For example, in the following code the *MakeDecision* expression calls the *GetDefaultDir* expression:

```
MakeDecision.function = DetermineUpdate
MakeDecision.defaultDir.expr = GetDefaultDir
<additional parameters>
GetDefaultDir.function = Get
<parameters>
```

When you call a second expression, the second expression will run, calculate a value, and return that value to the parameter that called it. For example, the *GetDefaultDir* expression might produce a value of c:/Program Files/IBM/. This value is then assigned to the *defaultDir* parameter, and the parameter's data type is automatically adjusted to match the data, as though the line read:

```
MakeDecision.defaultDir.path = c:/Program Files/IBM/
```

Parameter Types

Parameter types are similar to data types in programming. They determine which type of data a parameter contains.

Each parameter can contain an array of that data type. For example, if you set a parameter to be of the *path* type, the parameter could contain an array of paths.

Deployment Server supports the following parameter types:

alias The parameter's value is an alias (similar to a variable). You can use any name to represent an alias. Use aliases to store the results of an expression so you can refer to the result later. For example:

```
GetApplicationVersion.result.alias = ApplicationVersion
```

For more information about aliases, see ″Using Aliases in Manifests″ .

expr The parameter is calling another expression. That expression is evaluated, and the result is then assigned to the parameter.

For more information, see ″Using Nested Expressions″ .

htmlParam
The parameter's value is a list of HTML parameters. These HTML parameters are automatically retrieved from the <param> tag of the <applet> tag that loaded the active applet.

For more information about the applet tag, see ″Adding the Applet Tag″ .

literal The parameter's value is a literal string.

path The parameter's value is the path to a file on the user's computer. The following rules apply to writing a path:

- Use the slash character (/) as a file separator. For example, c:/Program Files/ACME/WebProgram/3.0/.
- Enclose aliases with the greater than and less than symbols. For example, <alias>.
- Use an asterisk (*) to denote a wildcard of any length. For example, c:/Program Files/ACME/WebProgram/*/.
  - Note that the asterisk only applies to one directory level. For example, c:/* does not represent the entire directory structure - it only represents the root directory.
- Use a question mark (?) to denote a single character wildcard. For example, c:/Program Files/ACME/WebProgram/3.?/.
- If the file or directory names in the path contain any reserved characters (/ \ * ? < >), escape the character with a backward slash (\ ). For example, c:/Program Files/ACME/WebProgram/3.0/WhatsThis\?/.

registry
The parameter's value is a registry key. The following rules apply to writing a registry key:

- Use the slash character (/) as a key separator. For example, HKEY_LOCAL_MACHINE/SOFTWARE/.

- Enclose aliases with the greater than and less than symbols. For example, HKEY_LOCAL_MACHINE/SOFTWARE/<WebProgramKey>/.
- Use an asterisk (*) to denote a wildcard of any length. For example, HKEY_LOCAL_MACHINE/SOFTWARE/*/.
  - Note that the asterisk only applies to one key level.
- Use a question mark (?) to denote a single character wildcard. For example, HKEY_LOCAL_MACHINE/SOFTWARE/WebProgram 5.?/.
- Enclose registry values within brackets ([ ]). For example, HKEY_LOCAL_MACHINE/SOFTWARE/Microsoft/Current Version/AppPaths/masqform.exe/[Path].
- To access the default value for a registry key, use [Default]. For example, HKEY_LOCAL_MACHINE/SOFTWARE/Microsoft/Current Version/AppPaths/run.exe/[Default].

Using Aliases in Manifests

Aliases are similar to variables in programming. They allow you to refer to specific information by a particular name. For example, if you determined the path to a particular file, you might want to store that information in an alias called *FilePath*. You could then use the name *FilePath* to refer to that information, just as you would use a variable name to refer to the value of a variable.

Aliases are created by assigning them to a parameter in a function. For example, if you used a *FindLocation* function to search for the location of a file, the result parameter would hold the name of the directory that contained that file. If you assign the result parameter to equal an alias name, then the result will be stored in that alias.

To do this, first you must set the parameter to be an alias type. For example:

```
MyFunction.result.alias = <value>
```

Next, you place an alias name in the value portion of the parameter. For example, to use an alias called *InstallApplicationDirectory*, you would write:

```
MyFunction.result.alias = InstallApplicationDirectory
```

When the function runs, the result of the function is stored in the alias *InstallApplicationDirectory*. You can then refer to that alias elsewhere in your manifest. When you do this, you must enclose the alias in the greater than and less than symbols (< >). This indicates that the manifest should use the value of the alias. For example, the following line uses the *FileVersion* alias we created earlier:

```
MyOtherFunction.argument1.literal = <InstallApplicationDirectory>
```

When Deployment Server processes this line, it will substitute the value of the alias. For example, if the directory that the application was installed into was called WebProgram 2.0.0, Deployment Server would substitute that value, as though the line read:

```
MyOtherFunction.argument1.literal = WebProgram 2.0.0
```

Predefined Aliases

Deployment Server uses a number of predefined aliases to control some behavior. These aliases are set at both the application and package level, as follows:

Predefined Application Aliases:

- **InstallApplication —** This alias determines whether the application should exist on the user's computer. If *true*, Deployment Server will process all of the package manifests belonging to that application. If *false*, Deployment Server will not check the packages, and will skip to the next application. This alias is automatically set by the *DetermineUpdate* function when it compares version numbers.
- **InstallApplicationDirectory —** This alias contains the path to the application. This is either the path in which the application is already installed, or the path to which the application should be installed. This alias is automatically set by the *DetermineUpdate* function, and is either the detected path (the *dir* parameter) or the default path (the *defaultDir* parameter).
- **InstalledApplicationVersion —** This alias contains the version number of the installed application. If no application is detected, this alias will be empty. This alias is automatically set by the *DetermineUpdate* function when it detects the version of the specified file (the *file* parameter).

Predefined Package Aliases:

- **InstallPackage —** This alias determines whether the package will be installed. If *true*, Deployment Server will install the package. If *false*, Deployment Server will not install the package. This alias is automatically set by the *DetermineUpdate* function when it compares version numbers.
- **InstallPackageDirectory —** This alias contains the path to the package. This is either the path in which the package is already installed, or the path to which the package should be installed. This alias is automatically set by the *DetermineUpdate* function, and is either the detected path (the *dir* parameter) or the default path (the *defaultDir* parameter).
- **InstalledPackageVersion —** This alias contains the version number of the installed package. If no package is detected, this alias will be empty. This alias is automatically set by the *DetermineUpdate* function when it detects the version of the specified file (the *file* parameter).

As indicated, these aliases are automatically set by the *DetermineUpdate* function. For the most part, these aliases are used internally by Deployment Server to control different aspects of the deployment process. However, the application aliases can also be useful when writing package manifests.

For example, if you have already detected the location of an application, and you know your package should be in the same directory, you might want to refer to the *InstallApplicationDirectory* alias in your package manifest. This saves you from having to read the correct path from the registry again.

For more information about the DetermineUpdate expression, see ″DetermineUpdate″.

Note: Application aliases will work both within an application manifest and within any of the package manifests for the same application. For example, you can refer to the InstallApplicationDirectory alias within a package manifest, and it will contain the same value as it did in the application manifest. In programming terms, application aliases are in scope in package manifests for that application.

Function Details

Deployment Server supports a number of functions that you can use to create expressions in your manifests. Each function is explained in detail in this section.

About the Result Parameter

In addition to the parameters listed for each function, all functions contain an optional *result* parameter. This parameter contains the return value of the function. For example, if you called a *GetFileVersion* function, this function would retrieve the version number of a particular file. The result parameter of the function would then store this value. This is useful if you want to assign the result of a function to an alias.

For more information about aliases, see “Using Aliases in Manifests” on page 33.

Function Descriptions

AndOperation

Use the *AndOperation* to determine whether a number of expressions are true and to combine the results of those expressions.

When called, the *AndOperation* calls any number of additional expressions. Each expression is evaluated in turn. If any expression returns empty or *false* result, the *AndOperation* ends (that is, no more expressions are called) and returns an empty result.

If none of the expressions returns an empty or *false* value, the *AndOperation* returns all of the results as an array. For example, if the first expression returns *c:/* and the second expression returns *Program Files/IBM*, then the *AndOperation* will return an array containing *c:/*, *Program Files/IBM*.

Note that the *AndOperation* may return multiple *true* values. For example, if the *AndOperation* called three expressions, and each expression returned *true*, the *AndOperation* would return an array containing *true*, *true*, *true*. The expression calling the *AndOperation* will interpret this as *true*.

This function uses the following parameters:

- **list** - A comma delimited list of expressions to evaluate. Note that this is a literal type.

For example, in the following manifest an *AndOperation* is used to call two *CheckIfFileExists* functions. If both files exist, the *AndOperation* will return *true*. If either file does not exist, the *AndOperation* will return an empty result.

```
CheckForFiles.function = AndOperation
CheckForFiles.list.literal = CheckForWebProgram, CheckForDesktopProgram
CheckForWebProgram.function = CheckIfFileExists
CheckForWebProgram.file.path = c:/Program Files/ACME/WebProgram/3.0/run.exe
CheckForDesktopProgram.function = CheckIfFileExists
CheckForDesktopProgram.file.path = c:/Program Files/ACME/DesktopProgram/
   3.0/run.exe
```

CompareVersion

Use *CompareVersion* to compare two version numbers with the operator of your choice. *CompareVersion* returns *true* if the comparison is true, and *false* if the comparison is false.

For example, if you tested whether 4.5.0.0 > 4.4.0.0 the result would be *true*.

This function uses the following parameters:

- **arg1 —** The first version number. This must be a four part version, such as 4.5.0.0.
- **arg2 —** The second version number. This must be a four part version, such as 4.5.0.0.
- **operation —** The operation to use for the comparison. Valid operators are: <, <=, =, =>, >, and !=.

For example, in the following manifest a *CompareVersion* function is used to determine whether the installed software is older than the current version. The *CompareVersion* function calls a *GetFileVersion* function to retrieve the version of the installed software, then compares this to the current version of 5.0.0.0.

```
IsInstallOlder.function = CompareVersion
IsInstallOlder.arg1.literal = 5.0.0.0
IsInstallOlder.operation.literal = >
IsInstallOlder.arg2.expr = GetInstalledVersion
GetInstalledVersion.function = GetFileVersion
GetInstalledVersion.file.path = c:/Program Files/ACME/WebProgram/3.0/run.exe
```

CheckIfFileExists

Use *CheckIfFileExists* to determine whether a file exists on the client computer.

When called, *CheckIfFileExists* will check the client computer for a specified file. If the file exists, the *CheckIfFileExists* will return *true*. If the file does not exist, *CheckIfFileExists* will return *false*.

This function uses the following parameters:

- **file —** The path and filename to check.

For example, in the following manifest an *AndOperation* is used to call two *CheckIfFileExists* functions. If both files exist, the *AndOperation* will return *true*. If either file does not exist, the *AndOperation* will return an empty result.

```
CheckForFiles.function = AndOperation
CheckForFiles.list.literal = CheckForWebProgram, CheckForDesktopProgram
CheckForWebProgram.function = CheckIfFileExists
CheckForWebProgram.file.path = c:/Program Files/ACME/WebProgram/3.0/run.exe
CheckForDesktopProgram.function = CheckIfFileExists
CheckForDesktopProgram.file.path = c:/Program Files/ACME/DesktopProgram/
   3.0/run.exe
```

CheckIfRegistryExists

Use *CheckIfRegistryExists* to determine whether a registry key exists on the client computer.

When called, *CheckIfRegistryExists* will check the client computer for a specified registry key. If the registry key exists, *CheckIfRegistryExists* will return *true*. If the registry key does not exist, *CheckIfRegistryExists* will return *false*.

This function uses the following parameters:

- **key —** The registry key to check.

For example, in the following manifest an *AndOperation* is used to call two *CheckIfRegistryExists* functions. If the registry keys exist, the *CheckIfRegistryExists* functions will both return true, and the *AndOperation* will return *true*. If either registry entry does not exist, the *AndOperation* will return an empty result.

```
   CheckForFiles.function = AndOperation
   CheckForFiles.list.literal = CheckForWebProgram, CheckForDesktopProgram
   CheckForWebProgram.function = CheckIfRegistryExists
   CheckForWebProgram.key.registry = HKEY_LOCAL_MACHINE/SOFTWARE/ACME/WebProgram/
      3.0.0/
   CheckForDesktopProgram.function = CheckIfRegistryExists
   CheckForDesktopProgram.key.registry = HKEY_LOCAL_MACHINE/SOFTWARE/ACME/
      DesktopProgram/3.0.0/
=======
   CheckForFiles.list.literal = CheckForWebProgram2, CheckForWebProgram1
   CheckForWebProgram2.function = CheckIfRegistryExists
   CheckForWebProgram2.key.registry = HKEY_LOCAL_MACHINE/SOFTWARE/ACME/WebProgram/
      2.0.0/
   CheckForWebProgram1.function = CheckIfRegistryExists
   CheckForWebProgram1.key.registry = HKEY_LOCAL_MACHINE/SOFTWARE/ACME/WebProgram/
      1.0.0/
>>>>>>> 1.4
```

DetermineUpdate

The *DetermineUpdate* function provides the logic that determines which packages Deployment Server will install. At the application level, *DetermineUpdate* decides whether to process the package manifests for that application. At the package level, *DetermineUpdate* decides whether to install each package.

DetermineUpdate attempts to locate a particular file on the user's computer, and to retrieve the version number of that file. The file's version number is then compared to the version number of the application or package being deployed, and a decision is made as follows:

- **Application —** If the version of the deployed application is greater than or equal to the version of the installed component, then *DetermineUpdate* returns *true* and the package manifests for that application will be processed.
- **Package —** If the version of the deployed package is greater than the version of the installed component, then *DetermineUpdate* returns *true* and the package will be installed.

If Deployment Server cannot find the specified path or file, then DetermineUpdate will return true, and either the package manifests will be processed or the package will be installed.

For example, suppose Deployment Server is deploying ACME WebProgram. Deployment Server would first check the user's registry to locate the path to the installed version of the WebProgram. Deployment Server would then retrieve the version number of the *run.exe* file in that directory. If the installed version was less than or equal to the version Deployment Server is deploying, then Deployment Server would proceed to the package level and check each package in turn. Any out of date or missing packages would then be installed.

This function uses the following parameters:

- **dir —** This is the directory in which the component is installed.

- **file** — This is the name of the file whose version will be retrieved.
- **defaultDir** — This is the directory to which the application will be installed if a previously installed path or file cannot be located.
- **version** — This is the four part version number of the application Deployment Server is deploying. For example, 5.0.0.0. If this parameter is not included, the version is taken from the *ApplicationVersion* or *PackageVersion* entry in the manifest.

Note: If you are deploying a file that does not have a version number, such as a text file, use a version number of 0.0.0.0. In this case, Deployment Server will detect whether the file already exists on the user's computer, and will install the file if it does not exist.

For example, to determine whether to install WebProgram 3.0, you would use the following expression:

```
MakeDecision.function = DetermineUpdate
MakeDecision.dir.registry = HKEY_LOCAL_MACHINE/SOFTWARE/Microsoft/CurrentVersion/
   App Paths/run.exe/[Default]
MakeDecision.defaultDir.literal = c:/Program Files/ACME/WebProgram/3.0
MakeDecision.file.literal = run.exe
MakeDecision.version.literal = 3.0.0.0
```

Get

Use *Get* to return a particular value or to determine whether a path or registry entry exists.

When called, the *Get* function will operate differently depending on whether the value parameter is a path or registry type:

- If the value parameter is not a path or registry type, the *Get* function will return the value in the value parameter.
- If the value parameter is a path or registry type, the *Get* function will evaluate to path or registry to determine if it exists on the client computer. If it does exist, the *Get* function will return the path or registry entry. If it does not exist, the *Get* function will check the onEmpty parameter for a default value. If there is a default value, the *Get* function will return it. If not, the *Get* function will return an empty result.

This function uses the following parameters:

- **value** — A value that the *Get* function returns.
- **onEmpty** — A default value. This value is used if the value parameter is empty, which may occur if the provided path or registry does not exist. This parameter is optional.

For example, the following manifest uses a *GetFileVersion* to call a *Get* functions. The *Get* function checks to determine whether a particular path exists. If it does, the *Get* function returns that path. Otherwise, the *Get* function returns a default path. The *GetFileVersion* then gets the version of the file at the returned path.

```
GetVersion.function = GetFileVersion
GetVersion.file.expr = GetPath
GetPath.function = Get
GetPath.value.path = c:/ACME/WebProgram/3.0/run.exe
GetPath.onEmpty.path = c:/Program Files/ACME/WebProgram/ 3.0/run.exe
```

GetFileVersion

Use *GetFileVersion* to get the version number of a file on the client computer. When called, this function reads the version number from a specific file and returns that value.

This function uses the following parameters:
- **file —** The complete path to the file.

For example, in the following manifest a *CompareVersion* function is used to determine whether the installed software is older than the current version. The *CompareVersion* function calls a *GetFileVersion* function to retrieve the version of the installed software, then compares this to the current version of 5.0.0.0.

```
IsInstallOlder.function = CompareVersion
IsInstallOlder.arg1.literal = 5.0.0.0
IsInstallOlder.operation.literal = >
IsInstallOlder.arg2.expr = GetInstalledVersion
GetInstalledVersion.function = GetFileVersion
GetInstalledVersion.file.path = c:/Program Files/ACME/WebProgram/3.0/run.exe
```

Note: The file's version number may not always match the product's version number. Be sure to check the file itself when determining which version number you want to look for.

NotOperation

Use the *NotOperation* if you want to evaluate an expression and return the opposite value.

When called, the *NotOperation* calls another expression that is evaluated. If the expression returns a non-empty value that is not *false*, then the *NotOperation* will return a value of *false*. If the expression returns an empty value or a value of *false*, then the *NotOperation* will return a value of *true*.

This function uses the following parameters:
- **name —** The name of the expression to evaluate. Note that this is a literal type.

For example, in the following manifest a *NotOperation* calls a *CheckIfFileExists* function. If the file exists, the *CheckIfFileExists* function returns *true*. The *NotOperation* then reverses this value, and returns *false*.

```
ReverseCheck.function = NotOperation
ReverseCheck.name.literal = CheckForWebProgram
CheckForWebProgram.function = CheckIfFileExists
CheckForWebProgram.file.path = c:/Program Files/ACME/WebProgram/3.0/run.exe
```

OrOperation

Use the *OrOperation* to determine if any one of a number of conditions are true, and to return the value of the true condition.

When called, the *OrOperation* calls any number of additional expressions. Each expression is evaluated in turn. If any expression returns a non-empty value that is not *false*, the *OrOperation* ends (that is, no more expressions are evaluated) and returns the value of that expression. If an expression returns an empty or *false* value, the *OrOperation* moves on to the next expression. If all expressions called return an empty or *false* value, the *OrOperation* returns the value of the last expression called (either empty or *false*).

For example, if the first expression returned an empty value, and the second expression returned *true*, then the *OrOperation* would return *true*.

This function uses the following parameters:
- **list —** A comma delimited list of expressions to evaluate. Note that this is a literal type.

For example, in the following manifest an *OrOperation* is used to call two *Set* functions. The first *Set* function checks to see if a particular path exists. If that path does not exist, the function returns empty, and the *OrOperation* calls the second *Set* function. The second *Set* function checks to see if a different path exists. If that path does not exist, the *Set* function returns a default value, which the *OrOperation* also returns.

```
SetPath.function = OrOperation
SetPath.list.literal = CheckWebProgramPathOne, CheckWebProgramPathTwo
CheckWebProgramPathOne.function = Get
CheckWebProgramPathOne.value.path = c:/Program Files/ACME/WebProgram/
CheckWebProgramPathTwo.function = Get
SetWebProgramPathTwo.value.path = c:/ACME/WebProgram/3.0/
SetWebProgramPathTwo.onEmpty.path = c:/Program Files/ACME/WebProgram/3.0/
```

RunOperation

Use the *RunOperation* to run a number of other expressions and combine the results.

When called, the *RunOperation* calls any number of additional expressions. Each expression is evaluated in turn, and the *RunOperation* returns all of the results as an array. For example, if the first expression returns *c:/* and the second expression returns *Program Files/IBM*, then the *RunOperation* will return an array containing *c:/, Program Files/IBM*.

This function uses the following parameters:
- **list —** A comma delimited list of expressions to evaluate.

For example, the following manifest uses a RunOperation to call two expressions. The first expression determines where the WebProgram is installed, and sets the WebProgramPath alias. The second expression gets the version of the WebProgram, and sets the WebProgramVersion alias. Finally, the RunOperation returns the union of the two, which might be *c:/Program Files/WebProgram/3.0/run.exe*, *2.5.2.0*. (Although this return value is not very useful, you can use the two aliases to refer to the specific values.)

```
DetermineVersion.function = RunOperation
DetermineVersion.list.literal = SetPath, GetVersion
SetPath.function = Set
SetPath.value.path = c:/ACME/WebProgram/3.0/run.exe
SetPath.onEmpty.path = c:/Program Files/ACME/WebProgram/3.0/run.exe
SetPath.result.alias = WebProgramPath
GetVersion.function = GetFileVersion
GetVersion.file.path = <WebProgramPath>
GetVersion.result.alias = WebProgramVersion
```

Set

Use *Set* to set the value of an alias or to determine whether a path or registry entry exists.

When called, the *Set* function will operate differently depending on whether the value parameter is a path or registry type:

- If the value parameter is not a path or registry type, the *Set* function will return the value in the value parameter.
- If the value parameter is a path or registry type, the *Set* function will evaluate to path or registry to determine if it exists on the client computer. If it does exist, the *Set* function will return the path or registry entry. If it does not exist, the *Set* function will check the onEmpty parameter for a default value. If there is a default value, the *Set* function will return it. If not, the *Set* function will return an empty result.

In either case, the *Set* function will also set the result parameter to the return value. This is useful if you want to assign that value to an alias.

This function uses the following parameters:

- **value** — A value that the *Set* function returns.
- **onEmpty** — A default value. This value is used if the value parameter is empty, which may occur if the provided path or registry does not exist. This parameter is optional.

For example, the following manifest uses a *RunOperation* to call two expressions. The first expression determines where the WebProgram is installed, and sets the *WebProgramPath* alias. The second expression gets the version of the WebProgram, and sets the *WebProgramVersion* alias. Finally, the *Set* operation returns an array containing both results. (Although this return value is not very useful, you can use the two aliases to refer to the specific values.)

```
DetermineVersion.function = RunOperation
DetermineVersion.list.literal = SetPath, GetVersion
SetPath.function = Set
SetPath.value.path = c:/ACME/WebProgram/3.0/run.exe
SetPath.onEmpty.path = c:/Program Files/ACME/WebProgram/3.0/run.exe
SetPath.result.alias = WebProgramPath
GetVersion.function = GetFileVersion
GetVersion.file.path = <WebProgramPath>
GetVersion.result.alias = WebProgramVersion
```

SetParent

Use *SetParent* to return the parent of a given path or registry entry. For example, if you had the path c:/Program Files/ACME/WebProgram/, *SetParent* would return the parent of that path, which is c:/Program Files/ACME/. If the path or registry entry ends in a specific filename or key name, *SetParent* will strip that name from the path or registry entry.

Note that unlike *Set*, *SetParent* does not check to see if the path or registry entry exists.

This function uses the following parameters:

- **value** — A path or registry entry.

For example, in the following manifest a *SetParent* function calls a *Get* function. The *Get* function checks to see if a particular path exists. If it does, then it returns that path, otherwise it returns a default path. The *SetParent* function then returns the parent of that path. (That is, if the *Get* function returned *c:/ACME/WebProgram/3.0/run.exe*, the *SetParent* function would return *c:/ACME/WebProgram/3.0/*.)

```
SetPath.function = SetParent
SetPath.value.expr = CheckFile
CheckFile.function = Get
CheckFile.value.path = c:/ACME/WebProgram/3.0/run.exe
CheckFile.onEmpty.path = c:/Program Files/ACME/WebProgram/3.0/run.exe
```

Adding the Manifests to Your File System

Once you have created your manifests, you must add them to your file system:

- **Master Manifest —** Copy the master manifest to the appropriate server directory.
- **Application Manifests —** Copy the application manifests to the appropriate application directories.
- **Package Manifests —** Copy the package manifests to the appropriate package directories. Be sure that the manifest matches the package file in that directory.

For more information about the file system, see “Using the Deployment Server File System” on page 15.

Setting Up Your Web Page to Use Deployment Server

You can set up any web page to use Deployment Server. Normally, you'll set up your web site in one of two ways:

- As a central deployment site where users can go to get the latest software.
- As specific pages that deploy applications relevant to those pages. For example, if your users need to complete a form as part of opening an account with your company, you may want to deploy the WebProgram during the account registration process.

To use Deployment Server, you need to include an applet tag in the appropriate web page. Since Deployment Server relies on a number of client-side components, such as Java support, you may also want your web pages to check for those components before running the applet.

Deployment Server is shipped with a template web site. You can use this web site to quickly set up a central deployment site, or as an example of how to set up a site. The web pages contain all of the logic necessary to detect the client-side components, and account for all possible scenarios when using Deployment Server.

Adding the Deployment Server Applet to a Web Page

You can add the Deployment Server applet to any web page using the standard applet tag. However, the applet tag and the applet you include will differ depending on whether your users are using Internet Explorer or a Mozilla browser.

You can account for this by setting up two different deployment pages, and using Javascript to detect your user's browser and direct them to the correct page.

To add an applet to a web page, you must:

- Add the applet tag to your HTML code.
- Copy the applet files to your web server.

Adding the Applet Tag

The applet tag has different attributes and parameters depending on whether your users are using Internet Explorer or Firefox. Each tag can support only one browser, and you will have to create different pages to support the different browsers.

The applet tag uses the following attributes

code This is the class in the applet that your browser runs. It must be:

```
com.PureEdge.ids.client.InstallApplet.class
```

height This is the height of the installation window the applet creates. You can assign any value.

width This is the width of the installation window the applet creates. You can assign any value.

archive

Include this attribute only for Firefox browsers. This attribute contains the relative path to the Firefox applet. The filename of the applet is:

```
IDS-NS.jar
```

For example, the applet tag for a Mozilla browser might look like this:

```
<applet code="com.PureEdge.ids.client.InstallApplet.class"
   height="200" width="500" archive="IDS-NS.jar">
```

The applet tag also requires the following parameters:

cabinets

Use this parameter only for Internet Explorer browsers. This parameter contains the relative path to the IE applets. You must always include the name of the applet, as shown:

```
IDS-IE.cab
```

cancel_url

This is the URL of the web page that is loaded if the user cancels the update.

fatal_url

This is the URL of the web page that is loaded if there is an error while performing the update.

security_denial_url

This is the URL of the web page that is loaded if the user refuses to give the applet the necessary permissions.

update_success_url

This is the URL of the web page that is loaded if the update is successful.

no_update_success_url

This is the URL of the web page that is loaded if no update is necessary.

locale This sets the locale of the applet. This must be the language code followed by the country code. For example:

```
fr_FR (french - France)
pt_BR (portuguese - Brazil)
zh_HK (traditional Chinese - Hong Kong)
```

Note: For a full list of supported languages and locales, see page 47.

For example, an applet tag for Internet Explorer might look like this:

```
<applet code="com.PureEdge.ids.client.InstallApplet.class"
   height="200" width="500">
   <param name="cabinets" value="IDS-IE.cab">
   <param name="bgcolor" value="#D6D6CE">
   <param name="cancel-url" value="http://www.sample.com/
      cancel.html">
   <param name="fatal_url" value="http://www.sample.com/
      fatal.html">
   <param name="security_denial_url" value="http://www.sample.com/
      denial.html">
   <param name="update_success_url" value="http://www.sample.com/
      success.html">
   <param name="no_update_success_url" value=
      "http://www.sample.com/noupdate.html">
<param name="locale" value="zh-HK">
</applet>
```

Copying the Applet Files

Before you can copy the applet files, you must first configure and sign the applets using the Signing Tool. For more information, see "Configuring Deployment Server" on page 9.

The Signing Tool produces two WAR files and three applet files:

- ServerIDS.war
- IDS.war
- IDS-IE.cab
- IDS-NS.jar

You must copy IDS-IE.cab to the location specified by the *cabinets* parameter of the applet tag, and you must copy IDS-NS.jar to the location specified by the *archive* attribute of the applet tag.

Note: Each applet tag is specific to either Internet Explorer or Firefox. If you are supporting both IE and Firefox, you will need to create multiple web pages with different applet tags. For more information, see "Adding the Applet Tag" on page 43.

About the Template Web Site

Deployment Server includes a template web site. This site include all of the functionality necessary to:

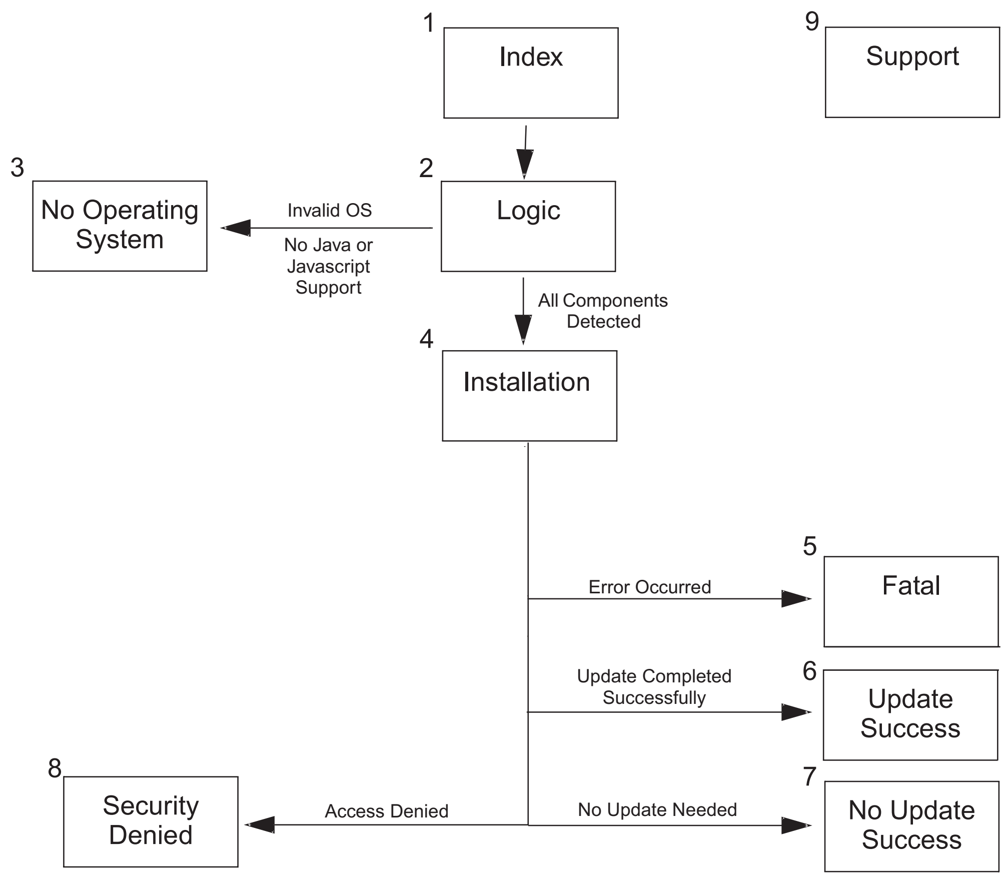
- Detect the user's configuration.
- Warn the user if they do not have the right configuration.
- Explain and launch the update process.
- Inform the user of the result of the update.

This site provides a working sample of the functionality you might want to include in your site. Optionally, you can quickly set up a Deployment Server site by using these pages and making some simple changes.

Architecture of the Template Web Site

The template web site uses a number of pages to step the user through the update process. The web site is a collection of JSP pages that use Javascript for detecting the user's configuration. Note that these pages were created using JSTL 1.0. For more information about JSTL, see The JSTL Expression Language.

The following diagram shows how the JSP pages are linked:



Each step corresponds to one or more JSP pages, and performs one or more actions, as listed:

Step	Action	JSP Page
1	This is the first page the user sees. It lists the end-user system requirements for using Deployment Server and links to step 2.	index.jsp
2	This is an invisible page that uses Javascript to determine if the user has the correct configuration: • If users do not have the correct configuration, they are sent to step 3. • If the user's system is configured correctly, this page determines which browser they are using, and links to the appropriate page in step 4.	logic.jsp
3	If the user systems are not configured correctly, they are directed here.	noSupport.jsp

Step	Action	JSP Page
4	This page contains the Deployment Server applet tag. The user's browser loads the Deployment Server applet, which begins the update process. The applet is specific to the browser in use, so there is a separate page for each type of browser: • Internet Explorer • Firefox Once the update is complete, the applet automatically loads the next page, depending on the results: • If an error occurs, the applet loads step 5. • If the update is successful, the applet loads step 6. • If no update is required, the applet loads step 7. • If the user refuses to give the applet the necessary security permissions, the applet loads step 8.	IEInstallation.jsp NSInstallation.jsp
5	If an error occurs during installation, the applet loads this page.	fatal.jsp
6	If the update is successful, the applet loads this page.	updateSuccess.jsp
7	If no update is required, the applet loads this page.	noUpdateSuccess.jsp
8	If the user refuses to give the applet the necessary security permissions, the applet loads this page.	securityDenied.jsp
9	This page provides information about getting help with Deployment Server. Users can link to this page from most of the other pages in the application.	support.jsp

Using the Template Web Site

If you use the template web site as a basis for your own Deployment Server web site, you will probably want to make some changes to the site to internationalize it and to maintain your own corporate image. Use the following JSP files to make this easier:

- **document_start.jsp —** This includes the common elements from the header of each page.
- **document_body.jsp —** This includes the common elements from the body of each page.
- **document_end.jsp —** This includes the common elements from the footer of each page.

If you edit these three files, you can change the basic look of the site to match your needs without having to edit the content or reconstruct the pages yourself.

You should make these changes to the appearance of the site before you configure and sign the applet. This will allow you to re-use your changes for each virtual server you want to set up.

Before you can edit the pages, you will have to extract them from the WAR file. You can then modify the files, and repackage them.

Note: These JSP pages have been localized to support a number of languages and locales. The version that the users see are displayed based on the language settings found on users' computers. These pages also allow users to select a language or locale from a drop-down list. This list includes:

- Chinese
  - Simplified Han (China and Singapore)
  - Traditional Han, Hong Kong
  - Traditional Han, Taiwan
- Croatian
- Czech
- Danish
- Dutch
- English
- French
- German
- Greek
- Hungarian
- Italian
- Japanese
- Korean
- Norewegian Bokmål
- Polish
- Portuguese
  - Brazil
  - Portugal
- Romanian
- Russian
- Slovak
- Slovenian
- Spanish
- Swedish
- Turkish

Modifying the Template Web Site

To modify the template web site:

1. Locate the IDS.WAR file that was installed with Deployment Server.
2. Rename the WAR file to IDS.zip.
3. Unzip the file.
4. Modify the .jsp pages as required.
5. Zip the .jsp pages, maintaining the same file list and structure as the original.
6. Rename the new zip file to IDS.war.

Once you have produced the new IDS.WAR file, you can use it in the configuration and signing process. For more information, see ″Configuring and Installing the Deployment Server Server and Applet″ .

Troubleshooting

Why did the Signing Tool fail to sign my certificate?
If you copy a signing certificate to your Mozilla-based browser certificate store, you must ensure that the certificate is trusted. In many cases, Mozilla-based browsers do not automatically mark the certificate as trusted. If the certificate is not trusted, the Signing Tool will fail.

I created a new virtual directory, but Deployment Server does not recognize it.
If you add a new virtual server directory to your Deployment Server file system, you may need to restart your servlet runner to make the Deployment Server recognize the new directory.

Deployment Server stopped creating log files.
If you want to modify the Deployment Server log file in any way (trimming, deleting, and so on), you must shut down the servlet runner first. If you do not, Deployment Server may stop logging, or other problems may occur.

Deployment Server does not recognize my deployment package.
WinZip 9 (and above) includes a Java compression method that is not supported by Deployment Server. If you create a deployment package using WinZip 9, Deployment Server may not be able to read it. Make sure you create deployment packages using versions of WinZip prior to version 9, or use other basic methods of creating zip packages.

Appendix. Notices

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Office 4360
One Rogers Street
Cambridge, MA 02142
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX
IBM
Workplace
Workplace Forms

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Index

A

alias parameter 32
aliases
- about 33
- InstallApplication alias 34
- InstallApplicationDirectory alias 34
- InstalledApplicationVersion alias 34
- InstalledPackageDirectory alias 34
- InstallPackage alias 34
- InstallPackageDirectory alias 34
- using aliases in manifests 33

AndOperation function 35
applet
- about 3
- adding the applet to your web page 43
- configuring the applet 9
- copying the applet files to your server 45
- dialogs shown by the applet 4
- security warning 4, 12
- signing the applet 13

application directories
- about 15
- setting up the application directories 17

application manifests
- about 21
- creating an application manifest 24

architecture
- architecture of the template web site 45
- IDS system architecture 3

C

certificates, code signing See code signing certificates 10
CheckIfFileExists function 36
CheckIfRegistryExists function 36
code signing certificates
- obtaining 10
- preparing 10

CompareVersion function 36
configuring
- configuring the applet 9
- configuring the server 9
- configuring the Signing Tool 11

conventions, document 1

D

dates, using date ranges in manifests 23
decisions, how manifests make decisions 21
DetermineUpdate function 37
directories
- about the application directories 15
- about the package directories 15
- about the root directory 15

directories *(continued)*
- about the server directories 15
- setting up the application directories 17
- setting up the package directories 17
- setting up the root directory 16
- setting up the server directories 17

document conventions 1
document_body.jsp 47
document_end.jsp 47
document_start.jsp 47

E

end-user experience 4
expressions
- about expressions 30
- adding an expression to a manifest 30
- using nested expressions 31

F

file system
- about 15
- adding manifests to your file system 42
- copying packages to the file system 20
- how IDS uses the file system 16
- setting up the file system 16

filenames, for manifests 22
functions
- about functions 30
- about the result parameter 35
- AndOperation 35
- calling a function in a manifest 30
- calling functions from other expressions 31
- CheckIfFileExists 36
- CheckIfRegistryExists 36
- CompareVersion 36
- DetermineUpdate 37
- function descriptions 35
- Get 38
- GetFileVersion 39
- NotOperation 39
- OrOperation 39
- RunOperation 40
- Set 40
- SetParent 41

G

Get function 38
GetFileVersion function 39

H

hierarchy, about the manifest hierarchy 21
htmlParam parameter 32

I

IDS
- IDS system architecture 3
- IDS system requirements 7
- overview of IDS 3
- setting up IDS 7

IDS applet See applet 9
IDS File System See file system 15
IDS server See server 9
InstallApplication alias 34
InstallApplicationDirectory alias 34
installation progress 4
InstalledApplicationVersion alias 34
InstalledPackageDirectory alias 34
InstallPackage alias 34
InstallPackageDirectory alias 34
Internet Deployment Server See IDS 3

J

JSP pages
- about 45
- document_body.jsp 47
- document_end.jsp 47
- document_start.jsp 47

K

key terms 2

M

manifest.properties 22
manifests
- about 21
- about the application manifests 21
- about the manifest hierarchy 21
- about the master manifest 21
- about the package manifests 21
- adding an expression to a manifest 30
- adding manifests to your file system 42
- calling a function in a manifest 30
- creating a master manifest 23
- creating a package manifest 26
- creating an application manifest 24
- creating manifests 22
- how manifests make decisions 21
- manifest filename 22
- using date ranges in manifests 23

master manifest
- about 21

master manifest *(continued)*
  creating a master manifest 23

N
NotOperation function 39

O
OrOperation function 39

P
package directories
  about 15
  setting up the package directories 17
package manifests
  about 21
  creating a package manifest 26
packages
  about 19
  copying packages to the file system 20
  creating your own packages 19
  using packages from IBM 19
parameter, result 35
parameters
  about 32
  alias paramater 32
  htmlParam parameter 32
  parameter types 32

R
requirements, system 7
result parameter 35
root directory
  about 15
  setting up the root directory 16
RunOperation function 40

S
security warning, applet 4, 12
server
  configuring the server 9
  copying the applet files to your server 45
server directories
  about 15
  setting up the server directories 17
servlet
  about 3
Set function 40
SetParent function 41
setting up
  setting up IDS 7
  setting up your web page to use IDS 43
signing the applet 13
Signing Tool
  configuring the Signing Tool 11
  using the Signing Tool 13
subroutines, using nested expressions 31
system architecture, IDS 3
system requirements 7

T
template web site
  about the template web site 45
  architecture of the template web site 45
  modifying the template web site 48
  using the template web site 47
terms, key 2
Tool, Signing See Signing Tool 11
types
  types of functions 35
  types of parameters 32

U
user experience 4

W
WAR files
  about the WAR files 14
  installing the WAR files 14
web page
  adding the applet to your web page 43
  setting up your web page to use IDS 43
web site
  about the template web site 45
  architecture of the template web site 45
  modifying the template web site 48
  using the template web site 47

Program Number: 5724-N08

Printed in USA

G325-2594-00