InterSystems CACHÉ®

Adding UNIX Installation Packages to a Caché Distribution

Version 2009.1
30 June 2009

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Adding UNIX Installation Packages to a Caché Distribution
Caché Version 2009.1 30 June 2009
Copyright © 2009 InterSystems Corporation
All rights reserved.

This book was assembled and formatted in Adobe Page Description Format (PDF) using tools and information from the following sources: Sun Microsystems, RenderX, Inc., Adobe Systems, and the World Wide Web Consortium at www.w3c.org. The primary document development tools were special-purpose XML-processing applications built by InterSystems using Caché and Java.

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Customer Support

Tel:	+1 617 621-0700
Fax:	+1 617 374-9391
Email:	support@InterSystems.com

Table of Contents

Adding UNIX Installation Packages to a Caché Distribution

This article describes how to add a new UNIX install package to an existing Caché distribution. It is presented in the form of a tutorial in which we create a simple package that copies additional files into the Caché instance directory.

Note: Because install packages are implemented through UNIX shell scripts, you can also write packages that perform much more complex operations.

Suppose we have written a Caché callout shared library (see the "Calling Out of Caché" chapter in *Using the Caché Callin and Callout Functions*) to connect to an imaging device named Foo9000. We compile this library as libfoo9000.so and want to install it with Caché. In addition, we want the installation to prompt users to provide the network server name for the device (Foo9000) to which we want the library to connect. This information will be stored in a configuration file in the Caché instance manager's directory *(install-dir*\mgr).

We start with an existing Caché kit:

```
~/kit:>ls
cinstall   cplatname docs   lgpl.txt NOTICE
copyright.pdf dist   kitlist LICENSE package
```

... and our compiled library (libfoo9000.so):

```
~/lib:>ls
libfoo9000.so
```

First, we need to choose a location in the kit to store our library, then copy the library to that location. By convention, platform-specific libraries go in dist/*package*/*platform* directories (for example, ~/kit/dist/foo9000/lnxsusex64):

```
~/kit:>cd dist
~/kit/dist:>mkdir foo9000
~/kit/dist:>cd foo9000
~/kit/dist/foo9000:>mkdir lnxsusex64
~/kit/dist/foo9000:>cd lnxsusex64
~/kit/dist/foo9000/lnxsusex64:>cp ~/lib/libfoo9000.so .
```

Next, we need to create the installation package directory and add the manifest.isc file (which describes the package) to it. In its simplest form, the manifest.isc file includes only the name of the package, which must be identical to the name of the package directory (foo9000).

```
~/kit/package:>mkdir foo9000
~/kit/package:>cd foo9000
~/kit/package/foo9000:>emacs manifest.isc
package: foo9000
```

Without any content the package does not do anything, but in this tutorial we want to do the following:

1. Prompt users for the name of the server hosting the Foo9000.
2. Save this information in a configuration file in the manager's directory ((*install-dir*\mgr).
3. Copy the library (libfoo9000.so) into the instance binary directory.

The package installer performs actions in phases, the most important of which are the following:

- “parameters” phase
- “install” phase

Note: Packages can contain Bourne shell scripts, with the same name as the phase, for each phase. The package installer runs the script for each package at the appropriate time during the phase. If your package script successfully completes its given phase, it returns an error code of 0 explicitly or implicitly via its final command; otherwise it returns a non-zero error code.

The “parameters” phase collects information necessary for the package's installation, typically by prompting users, and should not make any permanent changes to the system. Users are typically given the opportunity to cancel the installation after the “parameters” phase; if they do so, the installation should have had no effect on their system.

The “install” phase modifies the system. During the install phase users should *not* be prompted for information because the install may be unattended or automated.

Some packages do not require information from users and, therefore, do not need a “parameters” script. If the script for a particular phase is not included in a package, no actions are performed for that package during the phase.

Here is our first attempt at a “parameters” script for the foo9000 package:

```
~/kit/package/foo9000:>emacs parameters
#!/bin/sh
echo "Please enter host name of the Foo9000 imaging server: "
read host
echo "Host $host entered."
```

If we try running this script, as follows, we see that it does indeed prompt us for the host name. which it records in the *host* variable:

```
~/kit/package/foo9000:>sh parameters
Please enter host name of the Foo9000 imaging server:
cupertino
Host cupertino entered.
```

However, what do we do with the *host* value once we've acquired it? When the script is finished running, it will be lost and unavailable when we need to write it to the configuration file during the “install” phase.

Note: Remember that we don not want to create the configuration file now because the “parameters” phase should have no effect on the user's system.

The package installer provides a convenient pair of functions – **Import** and **Export** – that let multiple phases and multiple packages share information. We can use these functions by including them in the parameters.include file through the usual shell script mechanism:

```
#!/bin/sh
. parameters.include
echo "Please enter host name of the Foo9000 imaging server: "
read host
echo "Host $host entered."
Export foo9000.host $host
```

The **Export** function takes the name of a parameter variable to export and its value, typically from a variable local to the script. The **Import** function works in reverse: the first argument is the local variable into which you want to import the previously exported value, and the second argument is the name of the parameter variable to which it was exported.

Note: By convention, parameter variables are given a name of *package name.local variable name* (for example, foo9000.host).

Since our “parameters” script now collects all the Foo9000 information needed to complete the installation, we can turn to writing the “install” script:

```
~/kit/package/foo9000:>emacs parameters
#!/bin/sh
. parameters.include
Import host foo9000.host
echo host=$host > ????/mgr/foo9000.cfg
cp ????/dist/foo9000/????/libfoo9000.so ????/bin
```

There are a few details (`????` in the preceding script) we need to provide:

- Where is the instance directory in which the install is being created?
- Where is the kit we're installing from?
- Which platform is being installed?

Although we could include these questions in the “parameters” script, that may confuse users because they already entered that information earlier in the install. Instead, we can import parameter variables from other packages that can provide the information we need; for a list of existing parameter variables, see the “Using the Caché Parameter File on UNIX and Linux” appendix of the *Caché Installation Guide*.

In order to use the parameter variables from a particular package, we must inform the package installer that our package (foo9000) depends on the other package and, therefore, our package must be processed later in each phase than the other package. We do this by adding “prerequisite” values to our package's manifest.isc file:

```
~/kit/package/foo9000:>emacs manifest.isc
package: foo9000
prerequisite: server_location
prerequisite: legacy_dist
prerequisite: platform_selection
```

Now we can import parameter variables from these packages and use them to complete our install script:

```
~/kit/package/foo9000:>emacs install
#!/bin/sh
. parameters.include
Import host foo9000.host
Import tgtdir "server_location.target_dir"
Import srcdir "legacy_dist.source_dir"
Import platform_family "platform_selection.platform_family"
echo host=$host > $tgtdir/mgr/foo9000.cfg
cp $srcdir/dist/foo9000/$platform_family/libfoo9000.so $tgtdir/bin
```

Our package (foo9000) is nearly complete. The final task is to add our package to the prerequisite list for an appropriate preexisting package. Then, to complete installation of that package, the package installer will process ours. In this case, we want our library to be installed and configured any time a Caché server is installed, so we add our new package to the “database_server” package's prerequisite list inside its manifest.isc file:

```
~/kit/package/database_server:>emacs manifest.isc
package: database_server
prerequisite: legacy_dist
prerequisite: platform_selection
prerequisite: server
prerequisite: server_location
prerequisite: upgrade
prerequisite: available_disk_space
prerequisite: posix_tools
...
prerequisite: isql
prerequisite: zlib
prerequisite: udp
prerequisite: bi
prerequisite: foo9000
```

As you can see, many packages are required to create a server installation, but now, when we run **cinstall**, our package (foo9000) is configured and installed:

```
~/kit:>sudo ./cinstall
Your system type is 'SuSE Linux Enterprise Server 10 (x64)'.
Currently defined instances:
Cache instance 'TRWTURBO'
directory: /home/woodfin/TRWTURBO
versionid: 2008.2.0.456.0
conf file: cache.cpf (SuperServer port = 1972, WebServer = 57785)
status:  crashed, last used Sat Sep 13 08:37:32 2008
Enter instance name: TRWPACK1
Do you want to create Cache instance 'TRWPACK1' ? Y
```

```
...
Please enter host name of the Foo9000 imaging server:
cupertino
Host cupertino entered.
...
Do you want to proceed with the installation ? Y
...
Installation completed successfully
~/TRWPACK1/bin:>ls libfoo*
libfoo9000.so
~/TRWPACK1/mgr:>cat foo9000.cfg
host=cupertino
```