
Wigwam 4

Quick Start Guide

April 2004

042503WQG-1

Contents

Initialize playpen

- Initialize playpen using existing Wigwam project 1
- Bootstrap a new Wigwam webserver project 1

Add a new package to the package archive

- Check out the central package-archive as a playpen 6
- Assemble package files 7
- Test your new package in another project 12

Override Wigwam project components

- Override any project file 14
- Override a package meta file 14
- Override package source 16

Provide in-project packages

Migrate to Wigwam 4

- Prepare playpen for migration 19
- Re-bootstrap project with Wigwam 4 20
- Construct in-project source as a package 20
- Restructure your package overrides 22
- Install packages for the project 25
- Rebuild your playpen 25
- Share a migrated project 25

Use packagectl commands

- Install packages in your playpen 29
- Upgrade packages in your playpen 29
- Remove/uninstall packages from your playpen 30
- View package information 30

DRAFT

1. Initialize playpen

This section describes how to initialize a playpen by checking out an existing Wigwam project from revision control.

1.1 Initialize playpen using existing Wigwam project

Most developers typically begin a project by checking out an existing project.

1. Check the project out of revision control. The revision control system creates the project directory during the checkout process.

TIP FOR CVS: *In CVS, you can use `cvsc checkout project_name`.*

2. Run `./autogen.sh` in the top directory of the project to build a local playpen and install all Wigwam programs and project packages.

```
$PLAYPEN_ROOT> ./autogen.sh
```

3. Source `setup-env` from your top level project directory so that your shell environment knows about the Wigwam executables. For example:

```
$PLAYPEN_ROOT> . ./setup-env
```

1.2 Bootstrap a new Wigwam webserver project

In some cases, a developer may wish to start from scratch to develop his/her own project. The developer creates a skeleton project, then builds upon that.

Run wigwam-bootstrap

1. Get the latest `wigwam-bootstrap` from the `wigwam-base` download area. You can get it from one of the following location:

<http://herbie.ddv.com:1025/~pmineiro/wigwam/wigwam-bootstrap>

2. Run **wigwam-bootstrap** from any location. For example:

```
% wigwam-bootstrap --project=myproject
```

3. When **wigwam-bootstrap** finishes running, you have a skeletal project. Switch to that new project directory. For example:

```
% cd myproject
```

4. Source **setup-env** from your top level project directory so that your shell environment knows about the Wigwam executables. For example:

```
% . ./setup-env
```

Install packages

1. Use **packagectl install** to install packages, along with any of those packages' dependencies, into your new project:

```
% packagectl install service_static_apache

wigwam-base-4.0.2-1 post-installing. Done.
mime-types-0.1-3 post-installing. Done.
expat-1.95.6-1 post-installing. Done.
apache-1.3.29-1 post-installing. Done.
servicectl-1.0-1 post-installing. Done.
service_static_apache-0.2-3 post-installing. Done.
```

2. Source **setup-env** again so that so that your shell environment knows about any new Wigwam executables.

Start services

1. If you installed any service packages, attempt to start those services. For example:

```
% servicectl start static_apache
Fatal: check-environment: service static_apache required environment variable
OUTSIDE_HOSTNAME not set
Fatal: static_apache 0.2-3 start: failed
```

2. If you get a message similar to the one shown above, then the service has some environment variables that you need to set before you can start it. To determine the environment variables that you need to set, look at the package's setting for

service_required_envars in the file in ext/packages/\$PACKAGE/\$VERSION/config.

```
% cd ./ext/packages/service_static_apache/0.2-3
% cat config
.
.
.
service_required_envars="LOCAL_VAR OUTSIDE_HOSTNAME OUTSIDE_PORT
STATIC_HTTPD_PORT SERVER_ADMIN LOG_DIR"
```

3. Set the service variables in an appropriate project's configuration file, etc/project.conf. For example:

```
% cat > etc/project.conf
OUTSIDE_HOSTNAME=${OUTSIDE_HOSTNAME="foo"}
OUTSIDE_PORT=${OUTSIDE_PORT="6969"}
STATIC_HTTPD_PORT=${STATIC_HTTPD_PORT="6969"}
SERVER_ADMIN=${SERVER_ADMIN="f@z.com"}
LOG_DIR=${LOG_DIR="$LOCAL_VAR"}
```

4. Start the services.

```
% systemctl start
Info: static_apache ping: checking pid file ... not running.
Info: static_apache start: success.
```

5. Telnet to the local host.

```
% telnet localhost 6969
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
HEAD / HTTP/1.0

HTTP/1.1 404 Not Found
Date: Fri, 26 Dec 2003 19:15:24 GMT
Server: Apache/1.3.29 (Unix)
Connection: close
Content-Type: text/html; charset=iso-8859-1

Connection closed by foreign host.
```

You now have a webserver running, albeit without content.

TIP. The static apache service, like many services, comes with a configuration template, in this case stored in `ext/etc/static_apache.conf.in`. This template sets the document root for the server to `www/htdocs`, which currently does not exist. You can create this directory to serve up content, if you wish. Alternatively, you can override this file by editing your own copy in `etc/static_apache.conf.in`. For example, you could change the document root to point to a directory that exists (you must restart the service for this change to take effect).

Set up in revision control

1. Instantiate your project in the revision control repository using the appropriate revision control command. The following example uses CVS:

```
% cvs import -m 'initial import' myproject start import
```

2. Next, stop the running services. For example:

```
% systemctl stop
Info: static_apache ping: checking pid file ... running.
Info: stopping static_apache ... success.
```

3. Move your local copy out of the way. For example:

```
% cd ..
% mv myproject myproject.orig
```

4. Check out the project from revision control. The following example uses CVS:

```
% cvs co myproject
```

5. Rebuild your project playpen by running `./autogen.sh` from your project directory (i.e., from `$PLAYPEN_ROOT`). For example:

```
% cd ./myproject
% ./autogen.sh
```

6. Source `setup-env` to make sure that all executables are available to the shell. For example:

```
% . ./setup-env
```

7. Start up project services. For example:

```
% servicectl start  
Info: static_apache ping: checking pid file ... not running.  
Info: static_apache start: success.
```

2. Add a new package to the package archive

When you set up a package that is meant to be reusable across projects, you need to set up the package in the context of a package archive. The most straightforward procedure for creating a new package is as follows:

- Check out the central package archive as a project playpen
- Create a directory for your package
- Assemble the package files within that directory
- Test the package to be sure it installs in a project
- Commit your package archive playpen

2.1 Check out the central package-archive as a playpen

When you set up a package that is meant to be reusable across projects, you need to set up that package in the context of a package archive. This procedure is the most straightforward way to accomplish this task, but you may wish to contact a Wigwam expert for shortcuts.

1. Check out the central package archive from revision control. For example, the central package archive might be in `/wigwam/central-package-archive/`

TIP FOR CVS: *In CVS, you can use a command such as `cvs checkout central_package_archive`.*

```
% cvs co wigwam_package_archive
U wigwam-package-archive/README
U wigwam-package-archive/README.bootstraping
U wigwam-package-archive/README.packaging.utils
U wigwam-package-archive/README.renaming-packages
U wigwam-package-archive/README.wigwam-base
U wigwam-package-archive/autogen.sh
U wigwam-package-archive/setup-env
U wigwam-package-archive/beta-packages/.cvsignore
U wigwam-package-archive/beta-packages/installed_contents
U wigwam-package-archive/beta-packages/style
and so on....
```

2. At the top directory, run `./autogen.sh` to install `wigwam-base` and all its packages in a local project playpen, in a directory named `$PLAYPEN_ROOT/ext/`

3. Run `./autogen.sh` in the top directory of the project to build a local playpen (i.e., `$PLAYPEN_ROOT`) for the project and to install the package-archive packages in your playpen.

```
% cd project_dir
% ./autogen.sh
```

TIP. The first time you run `autogen.sh` on this checked-out archive, it may take awhile.

4. Source `setup-env` from your top level project directory so that your shell environment knows about executables that you'll be using from the `wigwam-packaging-utils` package. For example:

```
% . ./setup-env
```

You should now be ready to set up your package in this package archive project playpen.

2.2 Assemble package files

You use the same procedures to create either a raw or a make-type package, but the contents of those packages differs.

Set up make-type package

Make packages use standard functionality for downloading, building, and installing the specified package. This section describes the files you usually need to provide in the package.

1. Go to the main package directory (typically `$PLAYPEN_ROOT/public`) and create the `$PACKAGE/$VERSION` subdirectory for your make-type package:

```
% cd $PLAYPEN_ROOT/public
% mkdir mymakepackage
% cd mymakepackage
% mkdir 1.1.2-1
% cd 1.1.2-1
```

You can look at the *Developer Guide* if you need to review package naming conventions.

2. Create package `type` file. Set up a `type` file that simply indicates the type of package as `make`. For example:

```
% echo make > type
```

3. Create `urls` file where you specify the URLs of upstream source files that should not be committed to revision control. For example:

```
% cat > urls
http://wigwam.corp.p4pnet.net:1027/mymakeprogram/mymakeprogram-1.1.2-1.tar.gz
```

4. Provide the upstream source file. This can have a filename format of `*.tar.gz`, `*.tar.bz2`, `*.tgz`, `*.jar`, `*.zip`, or some other form. This is the source code that gets unpacked and installed when the package is installed into another project.
5. Create `dep` file for dependencies. For example:

```
% cat > dep
glib any
libpng [1.0.8-2,]
zlib [1.1.3,]
freetype
libjpeg
```

6. Set up options file. Make-type packages use an `options` file to set certain variables for the package. For more information about options files, you can look at the *Developer Guide* appendices. At minimum, the options file should contain `configure_style`. For example:

```
% cat > options
configure_style=perl
```

7. Create `files` file for package. Set up a list of all files that must be retrieved from the package archive for this package. If a source code file is included in the package itself, list that file as well.

```
% cat > files
dep
files
options
type
urls
mymakeprogram-1.1.2-1.tar.gz
```

8. Provide `md5sums` for all of the files listed in the `$PACKAGE/$VERSION/files`. Include the list of `md5sums` in the `$PACKAGE/$VERSION/md5sums` file. To create the `md5sums` file, use one of the generic utilities such as the Linux utility `md5sum`, which uses the syntax `md5sum FILENAME`. There are also other tools freely available for calculating `md5sums`.

Set up a raw-type package

Raw packages use standard functionality for downloading, building, and installing the specified package. This section describes the files you usually need to provide in the package.

1. Go to the main package directory (typically `$PLAYPEN_ROOT/public`) and create the `$PACKAGE/$VERSION` subdirectory for your raw-type package:

```
% cd $PLAYPEN_ROOT/public
% mkdir myrawpackage
% cd myrawpackage
% mkdir 1.1.2-1
% cd 1.1.2-1
```

If you need more information about package and version naming conventions, look at the *Developer Guide*.

2. Create package `type` file. Set up a `type` file that simply indicates the type of package as `raw`. For example:

```
% echo raw > type
```

3. Create `urls` file where you specify the URLs of upstream source files that should not be committed to revision control. For example:

```
% cat > urls
http://wigwam.corp.p4pnet.net:1027/myrawprogram/myrawprogram-1.1.2-1.jar
```

4. Provide the upstream source file. This can have a filename format of `*.tar.gz`, `*.tar.bz2`, `*.tgz`, `*.jar`, `*.zip`, or some other form. This is the source code that gets unpacked and installed when the package is installed into another project.
5. Create `dep` file for dependencies. For example:

```
% cat > dep
aspectj latest
nekohtml latest
servletapi
httpunit latest
commons-httpclient-bin latest
commons-logging-bin
junit latest
```

6. Provide `profile` file to set up environment variables for the package. For example:

```
% cat > profile
% CLASSPATH=${CLASSPATH}:${PLAYPEN_ROOT}/ext/lib/myrawprogram-1.1.2-1.jar
%
% etc.....
```

7. Provide `build` file to set up a build script for the package. For example:

```
% cat > build
exit 0
```

8. Provide `install` file to set up an install script for the package. For example:

```
% cat > install
cd $EXT_PKGBUILDDIR || {
    echo Failed to cd to $EXT_PKGBUILDDIR 1>&2
    exit 1
}

cp myrawprogram-1.1.2-1.jar $PLAYPEN_ROOT/ext/lib/.

test -d $PLAYPEN_ROOT/ext/etc/profile || mkdir \
$PLAYPEN_ROOT/ext/etc/profile
cp $EXT_PACKAGEDIR/${EXT_PACKAGE}-${EXT_VERSION}.profile \
$PLAYPEN_ROOT/ext/etc/profile
```

9. Create `files` file for package. Set up a list of all files that must be retrieved from the package archive for this package. If a source code file is included in the package itself, list that file as well.

```
% cat > files
build
dep
files
install
profile
type
urls
myrawprogram-1.1.2-1.jar
```

10. Provide `md5sums` for all of the files listed in the `$PACKAGE/$VERSION/files`. Include the list of `md5sums` in the `$PACKAGE/$VERSION/md5sums` file. To create the `md5sums` file, use one of the generic utilities such as the Linux utility `md5sum`, which uses the syntax `md5sum FILENAME`. There are also other tools freely available for calculating `md5sums`.

Set up a service-type package

You can package programs so that they may be run and managed as services within the project. Typically, service packages are based upon and dependent on another raw or make package that is also installed in the project.

1. Go to the main package directory (typically `$PLAYPEN_ROOT/public`) and create the `$PACKAGE/$VERSION` subdirectory for your raw-type package:

```
% cd $PLAYPEN_ROOT/public
% mkdir service_xyz
% cd service_xyz
% mkdir 1.1-1
% cd 1.1-1
```

If you need more information about package and version naming conventions, refer to the *Developer Guide*.

2. Create `config` file to configure the service so that, at runtime, it can be managed separately from the project itself. For example:

```
% cat > config
service_program=xyz
service_run_name=xyz
service_required_envvars="LOCAL_VAR XYZ_PORT XYZ_HOST"
```

3. Create `dep` file for dependencies. The dependencies indicate the packages that must be installed before your service package can be installed. For example:

```
% echo xyz > dep
```

4. Create `services` file. Set up a `services` file that simply indicates the program that this package enables as a service. For example:

```
% echo xyz > services
```

5. Create package `type` file. Set up a `type` file that simply indicates the type of package as raw. For example:

```
% echo service > type
```

6. Create `files` file for package. Set up a list of all files that must be retrieved from the package archive for this package. If a source code file is included in the package itself, list that file as well.

```
% cat > files
config
dep
files
services
type
```

7. Provide `md5sums` for all of the files listed in the `$PACKAGE/$VERSION/files`. Include the list of `md5sums` in the `$PACKAGE/$VERSION/md5sums` file. To create the `md5sums` file, use one of the generic utilities such as the Linux utility `md5sum`, which uses the syntax `md5sum FILENAME`. There are also other tools freely available for calculating `md5sums`.

2.3 Test your new package in another project

Once you've assembled your package files, you need to rebuild your package archive project `playpen`. In addition, before you commit to revision control, you should test your package in a separate project `playpen`. At minimum, you should ensure that the package can be installed into another Wigwam project.

1. Run `./autogen.sh` in the top directory of your package archive `playpen`. In package archive projects, running this program incorporates your new package into the archives and completes internal, archive-specific functions.

```
% cd $PLAYPEN_ROOT (for archive project)
% ./autogen.sh
```

2. Reference your package archive `playpen` in a separate project `playpen`. To do this, override the separate project's `package-archives` file and add your archive `playpen` to that file. For example:

```
% cd $PLAYPEN_ROOT (for separate project)
% cp ext/etc/package-archives etc/package-archives.local
% cat >> etc/package-archive.local
archive_${PLAYPEN_ROOT}_as_url
```

3. Install your new package in this separate project. For example:

```
% packagectl install mypackage 1.1.2-1  
mypackage 1.1.2-1 post-installing. Done.  
dependentpackage1 1.1.1-1 post-installing. Done.  
dependentpackage2 2.5-1 post-installing. Done.  
and so on...
```

If your package installs correctly, you can commit your entire package archive playpen to revision control. If it does not install correctly, you can refer to `ext/build/logs/$PACKAGE` to determine what went wrong.

NOTE. *You can look at the Developer Guide for more information about packagectl commands.*

3. Override Wigwam project components

This section provides quick start procedures for overriding individual files, files for an installed package, and an entire installed package.

3.1 Override any project file

For any file *X* used by wigwam, the file *X.local*, if it exists, acts like a replacement for *X*. Wigwam treats **.local* files specially. Wigwam sees any **.local* file as the developer's override for any particular file in the project.

Note that any file with the *.local* appended to it does not get checked into revision control; hence, it does not get communicated to other users of a project.

1. Go to the `$PLAYPEN_ROOT/` directory and copy the contents of *filename* to *filename.local*. For example:

```
% cd $PLAYPEN_ROOT
% cp etc/project.conf etc/project.conf.local
```

2. Modify the contents of *filename.local*.

TIP. You may override any Wigwam project file in the manner shown above. Some files, however, may cause unexpected behavior if you override them with a *.local* version. These files include `project-packages`, `in-project-packages`, `package-list`, and `installed-packages`.

3.2 Override a package meta file

If you're overriding a file in an installed package, and you want the file to override any current or future version of that same file within the archived package, use the procedure described here.

1. Go to the `$PLAYPEN_ROOT/packages` directory and create the directory where you'll place your override file. Use the format `$PACKAGE-$VERSION` to name your directory. For example:

```
% cd $PLAYPEN_ROOT/packages
% mkdir pkgX
```

2. Go to the `$PLAYPEN_ROOT` directory and copy the installed package file that you want to override to your new override directory location. For example:

```
% cd $PLAYPEN_ROOT
% cp ext/packages/pkgX/1.0-4/profile packages/pkgX/1.0-4/profile
```

3. If you want to provide a new file that's not already in the installed package, go to your new directory and provide a new file (i.e., a file not already installed in the package. For example:

```
% cd $PLAYPEN_ROOT/packages/pkgX
cat > build
exit 0
```

NOTE. Note that the `$PLAYPEN_ROOT/ext/packages/ subdirectories` have a `$PACKAGE/$VERSION/` format while the `$PLAYPEN_ROOT/packages/ subdirectories` have a `$PACKAGE` format.

4. Modify the contents of the file to suit your needs.
5. Reinstall the current version of `$PACKAGE`. For example:

```
% packagectl reinstall pkgX
```

6. You may now wish to:
 - a. Commit your playpen to revision control so that other project developers have access to the new override file.
 - b. Update your playpen
 - c. Go to `$PLAYPEN_ROOT`
 - d. Run `autogen.sh` to rebuild your playpen as necessary. The following example uses CVS:

```
% cvs update -d -P
% cd $PLAYPEN_ROOT
% ./autogen.sh
```

NOTE. When it accesses package files, Wigwam looks first in the `$PLAYPEN_ROOT/packages/$PACKAGE/` before it accesses `$PLAYPEN_ROOT/ext/packages/$PACKAGE/$VERSION/`. If both directories have the same filenames, Wigwam uses the first one that it finds (i.e., it uses the override).

3.3 Override package source

If you're overriding a file in an installed package, and you want the file to override any current or future version of that same file within the archived package, use the procedure described here.

1. Go to the `$PLAYPEN_ROOT/packages` directory and create the directory where you'll place your override file. For example:

```
% cd $PLAYPEN_ROOT/packages
% mkdir pkgX
```

2. Go to the `$PLAYPEN_ROOT/` directory and do either of the following:

- Copy the installed source file that you want to override to your new override location. For example:

```
% cd $PLAYPEN_ROOT/
% cp ext/src/pkgX/1.0-4/*.* src/pkgX/*.*
```

- OR, go to your new `$PLAYPEN_ROOT/src/$PACKAGE` directory and provide a completely different source file.

```
% cd $PLAYPEN_ROOT/packages/pkgX
cat > my_src
"source code here"
```

NOTE. Note that the `$PLAYPEN_ROOT/ext/src/` subdirectories have a `$PACKAGE/$VERSION/` format while the `$PLAYPEN_ROOT/src/` subdirectories have a `$PACKAGE` format. When it accesses package files, Wigwam looks first in the `$PLAYPEN_ROOT/src/$PACKAGE/` before it accesses `$PLAYPEN_ROOT/ext/src/$PACKAGE/$VERSION/`. If both directories have the same filenames, Wigwam uses the first one that it finds (i.e., it uses the override).

4. Provide in-project packages

This section describes how to provide packages that are intended only for the scope of the project and are not checked into version control.

Many projects need to include custom project-specific software. The examples below create a hello world application as in-project source code.

1. Create a `src` directory with a source code file. For example:

```
% cd $PLAYPEN_ROOT
% mkdir src
% cd src
% mkdir hello-world-local
% cd hello-world-local
% cat > hello-world.c
int main (void) { write (1, "hello world\n", 12); exit (0); }
% mkdir hello-world-local
% cat > Makefile
all: hello-world

install: hello-world
    cp hello-world ${PLAYPEN_ROOT}/ext/bin
```

2. Create a `packages` directory with package meta files. For example:

```
% cd $PLAYPEN_ROOT
% mkdir packages
% cd packages
% mkdir hello-world-local
% cd hello-world-local
% cat > options
configure_style="none"
% cat > type
% cat > files
options
type
files
```

3. Provide `md5sums` for all of the files listed in the `$PACKAGE/$VERSION/files`. Include the list of `md5sums` in the `$PACKAGE/$VERSION/md5sums` file. To create the `md5sums` file, use one of the generic utilities such as the Linux utility `md5sum`, which uses the syntax `md5sum FILENAME`. There are also other tools freely available for calculating `md5sums`.

4. Test your package to make sure that it installs, runs, and uninstalls. For example:

```
% packagectl install hello-world
hello-world-local post-installing. Done.
% hello-world
hello-world
% packagectl uninstall hello-world
hello-world-local uninstalling. Done.
% hello-world
zsh: command not found: hello-world
```

5. List your in-project package in the file `etc/in-project-packages`. By doing this, you set it up so that it installs each time you update the playpen (i.e., each time you run `./auto-gen.sh`). You can also use the `packagectl update-packages` to install the package. For example:

```
% cd $PLAYPEN_ROOT
% cat > etc/in-project-packages
hello-world
% packagectl update-packages
hello-world-local post-installing. Done.
```

5. Migrate to Wigwam 4

This section provides information on how migrate either your project, package, or package files so that they conform to Wigwam 4.

5.1 Prepare playpen for migration

Before you actually begin migrating your existing project to be Wigwam 4 compliant, do the following:

1. Modify the contents of the `$PLAYPEN_ROOT/etc/package-archives` so that it has both the Wigwam 3 package archives as well as the Wigwam 4 archives. List the Wigwam 4 archives *before* the Wigwam 3 archives.
2. Move the `project-packages` file aside. For example:

```
% mv etc/project-packages etc/project-packages.save
```

3. For now, remove the `ext/` directory. It will get rebuilt automatically later on. For example:

```
% cd $PLAYPEN_ROOT
% rm -rf ext
```

4. Remove `setup-env` from the revision control repository. The following example shows how you'd do this using CVS:

```
% cvs remove -f setup-env
% cvs commit -m 'no longer necessary' setup-env
```

5. Remove the Wigwam 3 `wigwam-bootstrap` binary.

```
% rm bin/wigwam-bootstrap
```

5.2 Re-bootstrap project with Wigwam 4

1. Download wigwam-bootstrap for version 4 and set permissions for it.

```
% cd $PLAYPEN_ROOT
% wget -O bin/wigwam-bootstrap \
'http://herbie.ddv.com/~pmineiro/wigwam/wigwam-bootstrap'
% chmod +x bin/wigwam-bootstrap
```

2. Commit wigwam-bootstrap to revision control, then run it from that location. The following example uses CVS:

```
% cvs commit -m 'upgrading to ww4' bin/wigwam-bootstrap
% bin/wigwam-bootstrap --from-cvs
```

3. Commit the Wigwam 4 executables of wigwam-bootstrap and autogen.sh to revision control. The following example uses CVS:

```
% cd $PLAYPEN_ROOT
% cvs commit -m 'upgrade to ww4' autogen.sh bin/wigwam-bootstrap
```

TIP FOR CVS: *Since setup-env is no longer revision-controlled in Wigwam 4, you don't commit that program.*

4. Move your project-packages file back. For example:

```
% cd $PLAYPEN_ROOT/etc
% mv etc/project-packages.save etc/project-packages
```

5.3 Construct in-project source as a package

For Wigwam 4, all project source needs to be Wigwam-packaged. If you've provided in-project source, you need to set it up as an in-project package. You can set in-project packages up with the version `local`.

Migrate source code to a package

1. Create a `/src/$PACKAGE-local` directory, where `$PACKAGE` is the name under which you're packaging your in-project source, and `local` is the version number for any in-project package. This example uses CVS for revision control:

```
% cd $PLAYPEN_ROOT/src
% mkdir my_ip_source_package-local
% cvs add my_ip_source_package-local
```

2. Move your in-project source to your newly-created `$PLAYPEN_ROOT/src/$PACKAGE-local/` directory.
3. **CVS-specific:** If you are using CVS and you want to preserve revision history for your files, go to `cvs-ssh` and do some manual file copying:

```
% ssh cvs-ssh
% cd /usr/local/cvsroot/my_ip_source/src
% tar cf - java | (cd my_ip_source_package-local; tar xvf -)
% tar cf - web | (cd my_ip_source_package-local; tar xvf -)
% cp build.xml my_ip_source_package-local/.
% logout
```

4. Remove the files that are no longer necessary. The following example uses CVS:

```
% cd $PLAYPEN_ROOT
% cvs upd -dP
% cvs remove -f project-build project-clean project-cleanbuild project-install bui
% cvs remove -f java web
% cvs commit -m 'moved to my_ip_source_package-local' java web
% rm -rf java web
% cvsq upd -dP
```

Set up meta files for in-project package

Once you've packaged your in-project source, you need to provide package meta data files too. This procedure creates meta data files for a raw-type in-project package.

1. Create a `$PLAYPEN_ROOT/packages/$PACKAGE-local` directory. The following example uses CVS for revision control.

```
% cd $PLAYPEN_ROOT/packages
% mkdir my_ip_source_package-local
% cvs add my_ip_source_package-local
% cd $PLAYPEN_ROOT/packages/my_ip_source_package-local
```

2. Create package meta data files in the `packages/$PACKAGE-local/` directory.

```
% echo raw > type
% cat > build
#!/bin/sh
ant all
^D
% cat > install
#!/bin/sh
ant install
^D
```

3. Create an `etc/in-project-packages` file and add your local package to that file. Do not include the version number (i.e., `local`). For example:

```
% cd $PLAYPEN_ROOT/etc
% echo my_ip_source_package > in-project-packages
```

TIP. If an `etc/in-project-packages` file already exists, be sure that you don't unintentionally overwrite the existing contents, if any.

Install your in-project package

Install your in-project package into the playpen. For example:

```
% packagectl install my_ip_source_package local
```

5.4 Restructure your package overrides

If you have package override files, you need to move them around a bit to conform to the Wigwam 4 structure.

Move files from package-overrides

This procedure describes how to migrate any files that are in `$PLAYPEN_ROOT/package-overrides`.

1. Create a `$PLAYPEN_ROOT/packages/$PACKAGE-$VERSION` directory. The following example uses CVS:

```
% cd $PLAYPEN_ROOT/packages
% mkdir my_pkg-1.1-2
% cvs add my_pkg-1.1-2
```

2. Move each override meta file from `$PLAYPEN_ROOT/package-overrides/` to your new `$PACKAGE-$VERSION/directory`.

```
% cd $PLAYPEN_ROOT/packages/my_pkg-1.1-2
% mv $PLAYPEN_ROOT/package-overrides/my_pkg* .
```

3. Rename each meta file to conform to the Wigwam 4 naming convention. For example:

```
% mv my_pkg-1.1-2.build build
% mv my_pkg-1.1-2.install install
```

4. Commit the files in to revision control. This example uses CVS:

```
% cvs commit -m "migrating my_pkg overrides to wigwam 4"
```

5. Go to `$PLAYPEN_ROOT/package-overrides/` and remove the files that are no longer necessary. The following example uses CVS:

```
% cd $PLAYPEN_ROOT/package-overrides
% cvs upd -dP
% cvs remove -f my_pkg-1.1-2.build
% cvs remove -f my_pkg-1.1-2.install
% cvs commit -m 'renamed and moved to my_pkg-1.1-2 directory in src'
% cvsq upd -dP
```

Move meta file overrides

If a package is installed in your `$PLAYPEN_ROOT/ext/packages/` directory and you provided an override for one or more of its meta files in your `$PLAYPEN_ROOT/packages/` directory, you need to move those files into package-specific directories and rename them using Wigwam 4 naming conventions.

1. Create a `$PLAYPEN_ROOT/packages/$PACKAGE-$VERSION/` directory. The following example shows how you would

```
% cd $PLAYPEN_ROOT/packages
% mkdir packageX-1.1-2
% cvs add packageX-1.1-2
```

2. Move each meta file for your package from `$PLAYPEN_ROOT/packages/` to the new `$PLAYPEN_ROOT/packages/$PACKAGE-$VERSION/` and rename the file. This example uses CVS for revision control:

```
% cd $PLAYPEN_ROOT/packages
% mv $PLAYPEN_ROOT/packages/packageX-1.1-2.options packageX-1.1-2/options
% cvs commit -m "migrating packageX overrides to wigwam 4"
```

3. Go to `$PLAYPEN_ROOT/packages/` and remove the files that are no longer necessary. The following example uses CVS for revision control:

```
% cd $PLAYPEN_ROOT/packages
% cvs upd -dP
% cvs remove -f packageX-1.1-2.options
% cvs commit -m 'renamed and moved to packageX-1.1-2 directory in src'
% cvsq upd -dP
```

5.5 Install packages for the project

Once you've prepared your playpen (see page 19); re-bootstrapped your project with Wigwam 4 (see page 20); constructed your in-project packages (see page 20); and reorganized your override files (see page 22), you can install all of the other packages that the project needs.

1. First, be sure that your `$PLAYPEN_ROOT/etc/package-archives` file lists a Wigwam 4 package URL (e.g, `http://herbie.ddv.com/~pmineiro/wigwam/wigwam-bootstrap`) *before* it lists any Wigwam 3 package archive URLs.
2. Install additional Wigwam programs in your playpen. These are the programs that are not automatically installed when you run `wigwam-bootstrap`. For example:

```
% packagectl install servicectl pubtool wigwam-packaging-utils
```

5.6 Rebuild your playpen

1. Run `autogen.sh` in the top directory of the project to rebuild the `ext/` directory in your playpen. For example:

```
% cd $PLAYPEN_ROOT
% ./autogen.sh
```

2. Source `setup-env` from your top level project directory so that your shell environment knows about the Wigwam executables that are available as a result of the packages you installed. For example:

```
% . ./setup-env
```

5.7 Share a migrated project

This section outlines how all developers that are working on a "project X" can now migrate their respective playpens to Wigwam 4.

If you're the one who migrated the project

Now that your `$PLAYPEN_ROOT` has been re-bootstrapped with Wigwam 4, and you've set up and/or installed all the packages in `in-project-packages` and `project-packages`, you can simply do the following:

1. Commit the revision-controlled directories to revision control so that the other project users know about them.
2. Inform other project users that "project X" has been migrated to Wigwam 4.

If you need to update your playpen to the migrated project

If you've been informed that a project you are using has been migrated from Wigwam 3 to Wigwam 4, you can migrate your playpen as follows:

Playpen has no novel source or overrides

If your local system's `$PLAYPEN_ROOT` has no novel files (i.e., files that have not been communicated to other users), and it has no playpen-specific overrides, you can use this procedure.

IMPORTANT. *If you have any in-project packages or overrides that you haven't already committed to revision control (before the project got migrated), you need to **skip** this procedure and go to *Playpen has novel source or overrides* on page 27.*

1. Move the entire "old" `$PLAYPEN_ROOT` out of the way on your local system.
2. Initialize a playpen by following the procedures in *Initialize by using an existing Wigwam project* on page 23.
3. Be sure that your `$PLAYPEN_ROOT/etc/package-archives` file lists a Wigwam 4 package URL (e.g, `http://herbie.ddv.com/~pmineiro/wigwam/wigwam-bootstrap`) *before* it lists any Wigwam 3 package archive URLs.
4. Run `autogen.sh` in the top directory to build the `ext/` directory in your playpen. For example:

```
% cd $PLAYPEN_ROOT
% ./autogen.sh
```

5. Source `setup-env` from your top level project directory. For example:

```
% . ./setup-env
```

Playpen has novel source or overrides

If your local system's `$PLAYPEN_ROOT` has some novel files or some package overrides that you have not yet committed to revision control, you need to move some information around before you reinitialize your playpen with Wigwam

1. If you have in-project source that hasn't yet been committed to revision control, use the procedures in *Construct in-project source as a package* on page 20; otherwise, skip to step 4.
2. Move the `etc/in-project-packages` file aside for now. For example:

```
% cd $PLAYPEN_ROOT/etc
% mv in-project-packages in-project-packages.save
```

3. Move the `etc/in-project-packages` file aside for now. For example:

```
% cd $PLAYPEN_ROOT/etc
% mv in-project-packages in-project-packages.save
```

4. If you have package or package meta file overrides that you haven't yet committed to version control, use the procedures in *Restructure your package overrides* on page 22; otherwise, skip to step 8.
5. Remove your `ext/` directory.
6. Initialize a playpen by following the procedures in *Initialize by using an existing Wigwam project* on page 23.
7. Move the `in-project-packages` file back into the project:
 - If there is already an `$PLAYPEN_ROOT/etc/in-project-packages` initialized from revision control, add the contents of your `in-project-packages.save` to that file, and delete your `in-project-packages.save` file. For example

```
% cd $PLAYPEN_ROOT/etc
% echo your_in_project_package_name >> in-project-packages
% rm in-project-packages.save
```

- If there is not already an `$PLAYPEN_ROOT/etc/in-project-packages` initialized from revision control, rename your `in-project-packages.save` as `in-project-packages` file.

```
% cd $PLAYPEN_ROOT/etc
% mv in-project-packages.save in-project-packages
```

8. Run `autogen.sh` in the top directory of the project to build the `ext/` directory in your playpen. For example:

```
% cd $PLAYPEN_ROOT
% ./autogen.sh
```

By running this, you:

- install and build all packages listed in `etc/project-packages/` and `etc/in-project-packages/` into your `ext/packages/` and `ext/src/` directories
 - create and/or update all Wigwam-specific directories and files
9. Source **setup-env** from your top level project directory so that your shell environment knows about the Wigwam executables that are available as a result of the packages you installed. For example:

```
% . ./setup-env
```

6. Use packagectl commands

This section provides quickstart procedures for installing and managing packages in an existing project.

6.1 Install packages in your playpen

You install packages using the **packagectl install** command from anywhere in `$PLAYPEN_ROOT`. The following example commands do the following:

- Install the latest version of a specific package, along with any of its dependencies.
- Install the latest version of a space-delimited list of packages
- Install a specified version of a package
- Install the latest version of a space-delimited list of packages, plus a specified version of the last package listed
- Load variables for the installed packages.

```
% packagectl install service_static_apache
% packagectl install mysql libgd perl-AnyData
% packagectl install python 1.5.2-1
% packagectl install perl-Apache-DBI perl-AxKit 1.6.1-2
% . ./setup-env
```

If **packagectl install** fails, you can check the error logs in `ext/build/logs/$PACKAGE` to determine what went wrong.

6.2 Upgrade packages in your playpen

You can upgrade to the latest versions of packages by using the **packagectl upgrade** command from anywhere in `$PLAYPEN_ROOT`. The following example commands do the following:

- Upgrade to the latest version of a specific package, along with any of its dependencies.
- Upgrade to the latest version of a space-delimited list of packages
- Upgrade to a specified version of a package
- Upgrade to the latest version of a space-delimited list of packages, plus a specified version of the last package listed

- Load variables for any possible new interactive variables.

```
% packagectl upgrade service_static_apache
% packagectl upgrade mysql libgd jre
% packagectl upgrade python 1.5.2-1
% packagectl upgrade perl-AnyData perl-Apache-DBI perl-AxKit 1.6.1-2
% ./setup-env
```

If **packagectl upgrade** fails, you can check the error logs in `ext/build/logs/$PACKAGE` to determine what went wrong.

6.3 Remove/uninstall packages from your playpen

You can uninstall packages from your playpen by using **packagectl uninstall** from anywhere in `$PLAYPEN_ROOT`. The following example commands do the following:

- Uninstall a package,.
- Uninstall a package and its dependencies, as a space-delimited list of packages.
- Load the environment to update shell variables

```
% packagectl uninstall perl-AnyData
% packagectl uninstall libgd libpng zlib freetype jpegsrc
% ./setup-env
```

6.4 View package information

This section provides examples of the commands that you can use to view package information.

1. View list of project packages: The required project packages are listed in `etc/project-packages`, which is a revision-controlled file. To view the project packages, you can simply use the UNIX **cat** command. For example:

```
% cd ./myproject
% cat etc/project-packages

wigwam-base 4.0.2-1
mime-types 0.1-3
expat 1.95.6-1
apache 1.3.29-1
servicectl 1.0.1-1
service_static_apache 0.2-3
...
```

2. View a list of installed packages: The required project packages are listed in `ext/etc/installed-packages`, which is a Wigwam-maintained, non-revision-controlled, playpen file. To view the installed packages, you can simply use the UNIX `cat` command. For example:

```
% cd ./myproject
% cat ext/etc/installed-packages

wigwam-base 4.0.2-1
mime-types 0.1-3
expat 1.95.6-1
apache 1.3.29-1
servicectl 1.0.1-1
service_static_apache 0.2-3
...
```

3. View package dependencies: To view package dependencies, use the `packagectl list-dependencies` command. For example:

```
% packagectl list-dependencies mypackage

mysql
DBI
Msql-Mysql-modules
perl-TimeDate
perl-gd
perl-Chart
perl-DB_File
apache
perl-MIME-tools
```

4. View package conflicts: To view package conflicts, use the `packagectl list-conflicts` command. For example:

```
% packagectl list-conflicts AxKit

perl-XML-LibXML 1.53
```

5. View package provides: To view package dependencies, use the `packagectl list-provides` command. For example:

```
% packagectl list-provides mysql4

mysql
database
```

