
2

Installing PostgreSQL

This chapter focuses on the requirements and steps involved in installing and configuring PostgreSQL. Many of the PostgreSQL capabilities are not enabled, by default. For example, support for the TCL language is a feature that must be explicitly requested during compile-time. As there are many other features that are not configured by default, we will cover the various flags and options you may use to enable them when compiling PostgreSQL. It is important that you carefully read through all the steps in this process before beginning installation.

This chapter will walk you through the installation steps on a Linux/Unix-style platform. Our installation platform is Linux, but these instructions should be compatible with most current Unix platforms.



Although PostgreSQL is capable of running on a Win32 platform, this book does not cover installation on Windows. The Win32 version of PostgreSQL requires the Cygwin environment and will not operate independently within Win32. Although Cygwin can be useful in many situations, the use of PostgreSQL in a Cygwin environment is not recommended.

Preparing for Installation

The installation of PostgreSQL is not difficult. However, there are some software requirements that you will need for the PostgreSQL compilation. All of the requirements—outside of the PostgreSQL source code—are GNU tools. If you are running Linux, there is a good chance that the tools are already installed. If you are running a BSD derivative, such as FreeBSD or MacOS X, you may have to download the tools.

If you find that you are missing any of the required components, first check your vendor's web site for the packages; otherwise, you may download them from <http://www.gnu.org>. It is also essential that you have enough disk space available to unpack and compile the source code on the filesystem to which you install. Disk-space requirements are discussed in the section titled "Disk Space."

Required Software Packages

You will most likely have some of the required software packages already installed on your system, if not all of them. These packages are as follows:

GNU make

GNU make is commonly known as *gmake* on non-GNU based systems, but is normally referred to as just *make* on GNU-based systems such as Linux. For consistency, we will refer to it as *gmake* throughout the majority of this book.

We recommend that you use at least *gmake* version 3.76.1 or higher when compiling PostgreSQL. To verify the existence and correct version number of *gmake*, type the command shown in Example 2-1.

Example 2-1: Verifying GNU make

```
$ gmake --version

GNU Make version 3.79.1, by Richard Stallman and Roland McGrath.
Built for i386-redhat-linux-gnu
Copyright (C) 1988, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 2000
    Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.

Report bugs to <bug-make@gnu.org>.
```

ISO/ANSI C Compiler

There are numerous ISO/ANSI C compilers available. The recommended compiler for PostgreSQL is the GNU C Compiler, although PostgreSQL has been known to build with compilers from different vendors. At the time of this writing, the most commonly distributed versions of GCC are 2.95 and 2.96 (Red-Hat Linux 7.x and Mandrake Linux 8.x). If you do not currently have GCC installed, you can download it by visiting the GNU website at <http://gcc.gnu.org>.

To check for the existence and version of GCC, enter the command shown in Example 2-2.

Example 2-2: Verifying GCC

```
$ gcc --version
2.95.3
```

GNU zip and tar

GNU zip is also called *gzip*. GNU zip is a compression utility that can compress as well as decompress files. All compressed, or *zipped*, files made with *gzip* have a *.gz* extension. You can test for the existence of the *gzip* program with the *gzip --version* command.

In addition to *gzip*, you will require a copy of *tar*, a utility used to group several files and directories into a single archive, as well as to unpack these archives onto the filesystem. An archived *tar* output file will typically contain a *.tar* extension. Files that are both archived by *tar* and compressed by *gzip* often have a *.tar.gz* compound extension, as is the case with the included PostgreSQL source distribution. You can test for *tar* with the *tar --version* command.

Example 2-3: Verifying gzip and tar

```
$ gzip --version
gzip 1.3
(1999-12-21)
Copyright 1999 Free Software Foundation
Copyright 1992-1993 Jean-loup Gailly
This program comes with ABSOLUTELY NO WARRANTY.
You may redistribute copies of this program
under the terms of the GNU General Public License.
For more information about these matters, see the file named COPYING.
Compilation options:
DIRENT_UTIME STDC_HEADERS HAVE_UNISTD_H HAVE_MEMORY_H HAVE_STRING_H
Written by Jean-loup Gailly.

$ tar --version
tar (GNU tar) 1.13.17
Copyright 2000 Free Software Foundation, Inc.
This program comes with NO WARRANTY, to the extent permitted by law.
You may redistribute it under the terms of the GNU General Public License;
see the file named COPYING for details.
Written by John Gilmore and Jay Fenlason.
```

Optional Packages

The following are some optional packages that you may want to have installed:

GNU Readline library

The GNU Readline library greatly increases the usability of *psql*, the standard PostgreSQL command-line console client. It adds all of the standard functionality of the GNU Readline library to the *psql* command line, such as being able to easily modify, edit, and retrieve command-history information with the

arrow keys and the ability to search the command history (also known as a *reverse-i-search*). If the Readline library is already installed on your system, the configuration process should automatically compile readline support with *psql*.



You may not need this package if you have NetBSD, as NetBSD has a *libedit* library, which provides Readline compatibility.

OpenSSL

OpenSSL is an Open Source implementation of the SSL/TLS protocols. OpenSSL is commonly used with utilities such as OpenSSH and Apache-SSL. PostgreSQL can make use of OpenSSL for encrypted connectivity between the *psql* client application and the PostgreSQL backend. You may also want to consider OpenSSL if you wish to use Stunnel. More information on OpenSSL is located at <http://www.openssl.org>. Installing and configuring Stunnel for use with PostgreSQL is discussed in Chapter 8, *Authentication and Encryption*.

Tcl/Tk

Tcl/Tk is a combination programming language and graphical toolkit. Although we don't cover the use of Tcl with PostgreSQL, we do cover the use of PgAccess, which is written in Tcl. If you wish to utilize the PgAccess application you will need to install the Tcl/Tk software. The website for Tcl/Tk is <http://tcl.activestate.com>.

Ant/JDK

The JDK is the Java Development Kit. It is required for Java development; hence, it is required by PostgreSQL if you wish to enable JDBC support. *Ant* is a Java-based build tool (somewhat like *gmake*) that is also required for JDBC support. The JDK can be downloaded from <http://java.sun.com/j2se/index.html>, and *Ant* can be downloaded from <http://jakarta.apache.org/ant/index.html>.

Disk Space

PostgreSQL does not require the extensive use of disk resources. In fact, in comparison to products such as Oracle, PostgreSQL could be considered fat free. However, PostgreSQL is a database, and as with any database, the requirements will grow as you continue to use PostgreSQL.

On an average Linux machine, you will need approximately 50 MB of hard-drive space to unpack the source and another 60 MB of hard drive space to compile the source. If you choose to run the regression tests, you will need an additional 30 MB. Depending on the configuration options you choose, PostgreSQL can take

anywhere from 8 to 15 MB of hard drive space once installed.



Remember that PostgreSQL's space requirements will grow as you use the system! Be sure to plan ahead for the amount of data you will be storing.

Trying to install on a system lacking in disk space is potentially dangerous! Before installing PostgreSQL, we recommend that you check your filesystem to be sure you have enough disk space in your intended installation partition (e.g., */usr/local*). If you have a GNU-based system, the *df* command should be at your disposal. Example 2-4 checks for free disk space, reported in 1k blocks.

Example 2-4: Verifying disk space

```
$ df -k
Filesystem      1k-blocks      Used Available Use% Mounted on
/dev/hda1       2355984      932660  1303644  42% /
/dev/hdb1       4142800     2133160  1799192  54% /home
/dev/hda6       1541680      272540  1190828  19% /usr/local
```

10 Steps to Installing PostgreSQL

PostgreSQL is included on the CD distributed with this book, but you may want to visit the PostgreSQL website to see if there is a newer version available. Many FTP sites make the source files for PostgreSQL available for download; a complete list of FTP mirrors can be found at <http://www.postgresql.org>.

Once you have connected to a PostgreSQL FTP mirror, you will see the stable releases located within a directory beginning with *v* followed by a version (such as *v7.1.3*). There should also be a symbolic link to the most recent stable release's directory called *latest/*.

Within this sub-directory is a list of package files. The complete PostgreSQL installation package is named *postgresql-[version].tar.gz* and should be the largest file in the list. The following sub-packages are also made available for download, and may be installed in any combination (though at least *base* is required):

postgresql-base-[version].tar.gz

The *base* package contains the bare minimum of source code required to build and run PostgreSQL.

postgresql-docs-[version].tar.gz

The *docs* package contains the PostgreSQL documentation in HTML format. Note that the PostgreSQL *man* pages are automatically installed with the *base* package.

postgresql-opt-[version].tar.gz

The *opt* package contains several optional extensions to PostgreSQL, such as the interfaces for C++ (*libpq++*), JDBC, ODBC, Perl, Python, and Tcl. It also contains the source required for multibyte support.

postgresql-test-[version].tar.gz

The *test* package contains the regression test suite. This package is required to run regression tests after compiling PostgreSQL.

Step 1: Creating the “postgres” User

Create a Unix user account to own and manage the PostgreSQL database files. Typically, this user is named *postgres*, but it can be named anything that you choose. For consistency throughout the book, the user *postgres* is considered the PostgreSQL root or superuser.

You will need to have root privileges to create the PostgreSQL superuser. On a Linux machine, you can use the command shown in Example 2-5 to add the *postgres* user.

Example 2-5: Adding the postgres user

```
$ su - -c "useradd postgres"
```



Do not try to use the *root* user as the PostgreSQL superuser. Doing so presents a large security hole.

Step 2: Installing the PostgreSQL Source Package

Once you have acquired the source for PostgreSQL, you should copy the PostgreSQL source package to a temporary compilation directory. This directory will be the path where you install and configure PostgreSQL. Within this path, you will extract the contents from the *tar.gz* file and proceed with installation.

Bear in mind that this will not be the location of the installed database files. This is a temporary location for configuration and compilation of the source package itself. If you have downloaded the PostgreSQL package from the Internet, it is probably not saved in your intended compilation directory (unless you explicitly chose to save there). A common convention for building source on Unix and Linux machines is to build within the */usr/local/src* path. You will most likely need root privileges to access this path. As such, the remaining examples in this chapter will involve the *root* user until otherwise specified.



If you are a user of a commercial Linux distribution, we strongly suggest that you verify whether or not you have PostgreSQL already installed. On RPM-based systems, such as SuSe, Mandrake, or Red-Hat, this can be done by using the following command: `rpm -qa | grep -i postgres`. If you do have PostgreSQL installed, there is a good chance that it is outdated. You will want to download and install the latest version of PostgreSQL available. An RPM installation of PostgreSQL will sometimes install scripts and programs such as *postmaster* and *psql* into globally accessible directories. This can cause conflicts with source-built versions, so before installing a new version, be sure to remove the RPM by using the `rpm -e <package name>` command.

To unpack PostgreSQL source code on a Linux system, first move (or copy, from the CD) the compressed source file into `/usr/local/src` (most people move their source files here to keep them separate from their home directories and/or other locations they may keep downloaded files). After moving it to the filesystem location where you wish to unpack it, use *tar* to unpack the source files. The commands to perform these actions are shown in Example 2-6.

Example 2-6: Unpacking the PostgreSQL source package

```
[root@host root]# cp postgresql-7.1.3.tar.gz /usr/local/src
[root@host root]# cd /usr/local/src
[root@host src]# tar -xzf postgresql-7.1.3.tar.gz
postgresql-7.1.3/
postgresql-7.1.3/ChangeLogs/
postgresql-7.1.3/ChangeLogs/ChangeLog-7.1-7.1.1
postgresql-7.1.3/ChangeLogs/ChangeLog-7.1RC1-to-7.1RC2
postgresql-7.1.3/ChangeLogs/ChangeLog-7.1RC2-to-7.1RC3
postgresql-7.1.3/ChangeLogs/ChangeLog-7.1RC3-to-7.1rc4
postgresql-7.1.3/ChangeLogs/ChangeLog-7.1beta1-to-7.1beta3
postgresql-7.1.3/ChangeLogs/ChangeLog-7.1beta3-to-7.1beta4
postgresql-7.1.3/ChangeLogs/ChangeLog-7.1beta4-to-7.1beta5
postgresql-7.1.3/ChangeLogs/ChangeLog-7.1beta5-to-7.1beta6
postgresql-7.1.3/ChangeLogs/ChangeLog-7.1beta6-7.1RC1
postgresql-7.1.3/ChangeLogs/ChangeLog-7.1rc4-7.1
postgresql-7.1.3/ChangeLogs/ChangeLog-7.1.1-7.1.2
postgresql-7.1.3/ChangeLogs/ChangeLog-7.1.2-7.1.3
postgresql-7.1.3/COPYRIGHT
[...]
[root@host root]# chown -R postgres.postgres postgresql-7.1.3
```

Notice the last command used in Example 2-6. The command is `chown -R postgres.postgres postgresql-7.1.3`. This command grants the ownership of the PostgreSQL source directory tree to *postgres*, which in turn enables you to compile PostgreSQL as the *postgres* user. Once the extraction and ownership change has

completed, you can switch to the *postgres* user to compile PostgreSQL, resulting in all compiled files automatically being owned by *postgres*.

For reference purposes, the following list is a description of the *tar* options used to extract the PostgreSQL source distribution:

x (extract)

tar will extract from the passed filename (as opposed to creating a new file).

v (verbose)

tar will print verbose output as files are extracted. You may omit this flag if you do not wish to see each file as it is unpacked.

z (zipped)

tar will use *gunzip* to decompress the source. This option assumes that you are using the GNU tools; other versions of *tar* may not support the *z* flag. In the event that you are not using the GNU tools, you will need to manually unzip the file using *gunzip* before you can unpack it with *tar*.

f (file)

tar will use the filename following the *f* parameter to determine from which file to extract. In our examples, this file is *postgresql-7.1.3.tar.gz*.

After you have completed the extraction of the files, switch to the *postgres* user and change into the newly created directory (e.g., */usr/local/src/postgres-7.1.3*). The remaining installation steps will take place in that directory.

Step 3: Configuring the Source Tree

Before compilation, you must configure the source, and specify installation options specific to your needs. This is done with the *configure* script.

The *configure* script is also used to check for software dependencies that are required to compile PostgreSQL. As *configure* checks for dependencies, it will create the necessary files for use with the *gmake* command.

To use the default installation script, issue the following command: *./configure*. To specify options that will enable certain non-default features, append the option to the *./configure* command. For a list of all the available configuration options, use *./configure --help*

There is a good chance that the default source configuration that *configure* uses will not be the setup you require. For a well-rounded PostgreSQL installation, we recommend you use at least the following options:

--with-CXX

Allows you to build C++ programs for use with PostgreSQL by building the *libpq++* library.

--enable-odbc

Allows you to connect to PostgreSQL with programs that have a compatible ODBC driver (such as Microsoft Access).

--enable-multibyte

Allows multibyte characters to be used, such as non-English language characters (e.g., Kanji).

--with-maxbackends=NUMBER

Sets *NUMBER* as the maximum number of allowed connections (32, by default).

You can also specify anything from the following complete list of configuration options:

--prefix=PREFIX

Specifies that files should be installed under the directory provided with *PREFIX*, instead of the default installation directory (*/usr/local/pgsql*).

--exec-prefix=EXEC-PREFIX

Specifies that architecture-dependent executable files should be installed under the directory supplied with *EXEC-PREFIX*.

--bindir=DIRECTORY

Specifies that user executable files (such as *psql*) should be installed into the directory supplied with *DIRECTORY*.

--datadir=DIRECTORY

Specifies that the database should install data files used by PostgreSQL's program suite (as well as sample configuration files) into the directory supplied with *DIRECTORY*. Note that the directory here is *not* used as an alternate database data directory; it is merely the directory where read-only files used by the program suite are installed.

--sysconfdir=DIRECTORY

Specifies that system configuration files should be installed into the directory supplied with *DIRECTORY*. By default, these are put into the *etc* folder within the specified base installation directory.

--libdir=DIRECTORY

Specifies that library files should be stored in the directory supplied with *DIRECTORY*. If you are running Linux, this directory should also be entered into the *ld.so.conf* file.

--includedir=DIRECTORY

Specifies that C and C++ header files should be installed into the directory supplied with *DIRECTORY*. By default, include files are stored in the *include* folder within the base installation directory.

--docdir=DIRECTORY

Specifies that documentation files should be installed into the directory supplied with *DIRECTORY*. This does not include PostgreSQL's *man* files.

--mandir=DIRECTORY

Specifies that *man* files should be installed into the directory supplied with *DIRECTORY*.

--with-includes=DIRECTORIES

Specifies that the colon-separated list of directories supplied with *DIRECTORIES* should be searched with the purpose of locating additional header files.

--with-libraries=DIRECTORIES

Specifies that the colon-separated list of directories supplied with *DIRECTORIES* should be searched with the purpose of locating additional libraries.

--enable-locale

Enables locale support. The use of locale support will incur a performance penalty and should only be enabled if you are not in an English-speaking location.

--enable-recode

Enables the use of the *recode* translation library.

--enable-multibyte

Enables multibyte encoding. Enabling this option allows the support of non-ASCII characters; this is most useful with languages such as Japanese, Korean, and Chinese, which all use nonstandard character encoding.

--with-pgport=NUMBER

Specifies that the the port number supplied with *NUMBER* should be used as the default port by PostgreSQL. This can be changed when starting the *postmaster* application.

--with-maxbackends=NUMBER

Sets *NUMBER* as the maximum number of allowed connections (32, by default).

--with-CXX

Specifies that the C++ interface library should be compiled during installation. You will need this library if you plan to develop C++ applications for use with PostgreSQL.

--with-perl

Specifies that the PostgreSQL Perl interface module should be compiled during installation. This module will need to be installed in a directory that is usually owned by *root*, so you will most likely need to be logged in as the *root* user to complete installation with this option chosen. This configuration option is only required if you plan to use the pl/Perl procedural language.

--with-python

Specifies that the PostgreSQL Python interface module should be compiled during installation. As with the *--with-perl* option, you will most likely need to log in as the *root* user to complete installation with this option. This option is only required if you plan to use the pl/Python procedural language.

--with-tcl

Specifies that Tcl support should be included in the installation. This option will install PostgreSQL applications and extensions that require Tcl, such as *pgaccess* (a popular graphical database client) and the pl/Tcl procedural language.

--without-tk

Specifies that Tcl support should be compiled without additional support for Tk, the graphical application tool kit. Using this option with the *--with-tcl* option specifies that PostgreSQL Tcl applications that require Tk (such as *pgtksh* and *pgaccess*) should not be installed.

--with-tclconfig=DIRECTORY, --with-tkconfig=DIRECTORY

Specifies that the Tcl or Tk (depending on the option) configuration file (either *tclConfig.sh* or *tkConfig.sh*) is located in the directory supplied with *DIRECTORY*, instead of the default directory. These two files are installed by Tcl/Tk, and the information within them is required by PostgreSQL's Tcl/Tk interface modules.

--enable-odbc

Enables support for ODBC.

--with-odbcinst=DIRECTORY

Specifies that the ODBC driver should look in the directory supplied with *DIRECTORY* for its *odbcinst.ini* file. By default, this file is held in the *etc* directory, which is located in the installation directory.

--with-krb4=DIRECTORY, --with-krb5=DIRECTORY

Enables support for the Kerberos authentication system. The use of Kerberos is not covered in this book.

--with-krb-srvnam=NAME

Specifies the name of the Kerberos service principal. By default, *postgres* is set as the service principal name.

--with-openssl=DIRECTORY

Enables the use of SSL to support encrypted database connections. To build support for SSL, OpenSSL must be configured correctly and installed in the directory supplied with *DIRECTORY*. This option is required if you plan on using the *stunnel* tool.

--with-java

Enables Java/JDBC support. The *Ant* and JDK packages are required for PostgreSQL to compile correctly with this feature enabled.

--enable-syslog

Enables the use of the *syslog* daemon for logging. You will need to specify that you wish to use *syslog* for logging at runtime if you wish to use it.

--enable-debug

Enables the compilation of all PostgreSQL libraries and applications with debugging symbols. This will slow down performance and increase binary file size, but the debugging symbols are useful for developers to help diagnose bugs and problems that can be encountered with PostgreSQL.

--enable-cassert

Enables assertion checking. This feature slows down performance and should be used only during development of the PostgreSQL system itself.

If you compile PostgreSQL and find that you are missing a feature, you can return to this step, reconfigure, and continue with the subsequent steps to build and install PostgreSQL. If you choose to come back to this step and reconfigure the PostgreSQL source before installing, be sure to use the *gmake clean* command from the top-level directory of the source tree (usually, */usr/local/src/postgresql-[version]*). This will remove any leftover object files and partially compiled files.

Step 4: Compiling the Source

After using the *configure* command, you may begin compiling the PostgreSQL source by entering the *gmake* command.



On Linux machines, you should be able to use *make* instead of *gmake*. BSD users should use *gnumake*.

Example 2-7: Compiling the source with GNU make

```
[postgres@host postgresql-7.1.3]# make
make -C doc all
make[1]: Entering directory /usr/local/src/postgresql-7.1.3/doc'
make[1]: Nothing to be done for all'.
make[1]: Leaving directory /usr/local/src/postgresql-7.1.3/doc'
make -C src all
make[1]: Entering directory /usr/local/src/postgresql-7.1.3/src'
make -C backend all
make[2]: Entering directory /usr/local/src/postgresql-7.1.3/src/backend'
make -C utils fmgroids.h
make[3]: Entering directory /usr/local/src/postgresql-7.1.3/src/backend/utils'
[...]
```

At this point, depending on the speed of your machine, you may want to get some coffee because the PostgreSQL compilation could take 10 minutes, an hour, or even more. After the compilation has finished, the following message should appear:

```
All of PostgreSQL is successfully made. Ready to install.
```

Step 5: Regression Testing

Regression tests are an optional but recommended step. The regression tests help verify that PostgreSQL will run as expected after you have compiled the source. The tests check tasks such as standard SQL operations, as well as extended capabilities of PostgreSQL. The regression tests can point out possible (but not necessarily probable) problems which may arise when running PostgreSQL.

If you decide you would like to run the regression tests, do so by using the following command: *make check*, as shown in Example 2-8.

Example 2-8: Making regression tests

```
[postgres@host postgresql-7.1.3]# make check
make -C doc all
make[1]: Entering directory /usr/local/src/postgresql-7.1.3/doc'
make[1]: Nothing to be done for all'.
make[1]: Leaving directory /usr/local/src/postgresql-7.1.3/doc'
[...]
```

The *make check* command will build a test installation of PostgreSQL within the source tree, and display a list of all the checks it is running. As each test completes, the success or failure will be reported. Items that fail the check will have a `failed` message printed, rather than the successful `ok` message. If any checks fail, *make check* will display output similar to that found in Example 2-9, though the number of tests failed may be higher on your system than the number in the example.

Example 2-9: Regression check output

```
=====
1 of 76 tests failed.
=====
```

The differences that caused some tests to fail can be viewed in the file `./regression.diffs`. A copy of the test summary that you see above is saved in the file `./regression.out`.

The files referenced in Example 2-9 (*regression.diffs* and *regression.out*) are placed within the source tree at *src/test/regress*. If the source tree is located in `/usr/local/src`, the full path to the directory files would be `/usr/local/src/postgresql-[version]/src/test/regress`.

The regression tests will not always pick up every possible error. This can be due to inconsistencies in locale settings (such as time zone support), or hardware-specific issues (such as floating-point results). As with any application, be sure to perform your own requirements testing while developing with PostgreSQL.



You cannot run the regression tests as the *root* user. Be sure to run *gmake check* as the *postgres* user.

Step 6: Installing Compiled Programs and Libraries

After you have configured and compiled the PostgreSQL source code, it is time to install the compiled libraries, binaries, and data files into a more appropriate home on the system. If you are upgrading from a previous version of PostgreSQL, be sure to back up your database before beginning this step. Information on performing PostgreSQL database backups can be found in Chapter 9, *Database Management*.

Installation of the compiled files is accomplished with the commands demonstrated in Example 2-10. When executed in the manner shown in Example 2-10, the *su* command temporarily logs you in as the *root* user to execute the required commands. You must have the *root* password to execute both of the commands shown in Example 2-10.



If you specified a non-default installation directory in Step 3, use the directory you specified instead of `/usr/local/pgsql`.

Example 2-10: The `gmake install` command

```
$ su -c "gmake install"
Password:
gmake -C doc install
gmake[1]: Entering directory /usr/local/src/postgresql-7.1.3/doc'
mkdir /usr/local/pgsql
mkdir /usr/local/pgsql/man
mkdir /usr/local/pgsql/doc
mkdir /usr/local/pgsql/doc/html
[...]
$ su -c "chown -R postgres.postgres /usr/local/pgsql"
Password:
```

The `su -c "gmake install"` command will install the freshly compiled source either into the directory structure you chose in Step 3 with the `--prefix` configuration option, or, if this was left unspecified, into the default directory of `/usr/local/pgsql`. The use of the `su -c "chown -R postgres.postgres /usr/local/pgsql"` command will ensure that the `postgres` user owns the PostgreSQL installation directories. Using the `su -c` command lets you save a step by only logging you in as the `root` user for the duration of the command's execution.

If you chose to configure the PostgreSQL source with the Perl or Python interface, but did not have root access, you can still install the interfaces manually. Use the commands demonstrated in Example 2-11 to install the Perl and Python modules manually.

Example 2-11: Installing Perl and Python modules manually

```
$ su -c "gmake -C src/interfaces/perl5 install"
Password:
Password:
gmake: Entering directory /usr/local/src/postgresql-7.1.3/src/interfaces/perl5'
perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Pg
gmake -f Makefile clean
[...]
$ su -c "gmake -C src/interfaces/python install"
Password:
gmake: Entering directory /usr/local/src/postgresql-7.1.3/src/interfaces/python'
sed -e 's,@libpq_srcdir@,../../../../src/interfaces/libpq,g' \
    -e 's,@libpq_builddir@,../../../../src/interfaces/libpq,g' \
    -e 's%@EXTRA_LIBS@% -lz -lcrypt -lresolv -lnsl -ldl -lm -lbsd -lreadline -ltermcap %g' \
    -e 's%@INCLUDES@%-I../../../../src/include%g' \
[...]

```

You may also want to install the header files for PostgreSQL. This is important, because the default installation will only install the header files for client application development. If you are going to be using some of PostgreSQL's advanced

functionality, such as user-defined functions or developing applications in C that use the *libpq* library, you will need the header files. To install the required header files, perform the commands demonstrated in Example 2-12.

Example 2-12: Installing all headers

```
$ su -c "gmake install-all-headers"
Password:
gmake -C src install-all-headers
gmake[1]: Entering directory /usr/local/src/postgresql-7.1.3/src'
gmake -C include install-all-headers
[...]
```

Step 7: Setting Environment Variables

The use of the PostgreSQL environment variables is not required. However, they are helpful when performing tasks within PostgreSQL, including starting and shutting down the *postmaster* processes. The environment variables that should be set are for the *man* pages and the *bin* directory. You can do so by adding the following statements into the */etc/profile* file. This should work for any *sh*-based shell, including bash and ksh.

```
PATH=$PATH:/usr/local/pgsql/bin
MANPATH=$MANPATH:/usr/local/pgsql/man
export PATH MANPATH
```



You must login to the system *after* the */etc/profile* file has had environment variables added to it in order for your shell to utilize them.

Depending on how your system handles shared libraries, you may need to inform the operating system of where your PostgreSQL shared libraries are located. Systems such as Linux, FreeBSD, NetBSD, OpenBSD, Irix, HP/UX, and Solaris will most likely not need to do this.

In a default installation, shared libraries will be located in */usr/local/pgsql/lib* (this may be different, depending on whether you changed it with the *-prefix* configuration option). One of the most common ways to accomplish this is to set the `LD_LIBRARY_PATH` environment variable to */usr/local/pgsql/lib*. See Example 2-13 for an example of doing this in Bourne-style shells and Example 2-14 for an example of doing this in *csb* and *tcsb*.

Example 2-13: Setting LD_LIBRARY_PATH in a bash shell

```
$ LD_LIBRARY_PATH=/usr/local/pgsql/lib
$ export LD_LIBRARY_PATH
```

Example 2-14: Setting LD_LIBRARY_PATH in csh and tcsh

```
$ setenv LD_LIBRARY_PATH /usr/local/pgsql/lib
```

Step 8: Initializing and Starting PostgreSQL

If you are logged in as the *root* user, instead of using the *su -c* command in the previous steps, you will now need to login as the *postgres* user you added in step 1. Once you are logged in as the *postgres* user, issue the command shown in Example 2-15.

Example 2-15: Initializing the database

```
$ /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
```

The *-D* option in the previous command is the location where the data will be stored. This location can also be set with the *PGDATA* environment variable. If you have set *PGDATA*, the *-D* option is unnecessary. If you would like to use a different directory to hold these data files, make sure the *postgres* user account can write to that directory. When you execute *initdb* you will see something similar to what is shown in Example 2-16.

Example 2-16: Output from initdb

```
$ /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
This database system will be initialized with username "postgres."
This user will own all the data files and must also own the server process.

Creating directory /usr/local/pgsql/data
Creating directory /usr/local/pgsql/data/base
Creating directory /usr/local/pgsql/data/global
Creating directory /usr/local/pgsql/data/pg_xlog
Creating template1 database in /usr/local/pgsql/data/base/1
DEBUG: database system was shut down at 2001-08-24 16:36:35 PDT
DEBUG: CheckPoint record at (0, 8)
DEBUG: Redo record at (0, 8); Undo record at (0, 8); Shutdown TRUE
DEBUG: NextTransactionId: 514; NextOid: 16384
DEBUG: database system is in production state
Creating global relations in /usr/local/pgsql/data/global
DEBUG: database system was shut down at 2001-08-24 16:36:38 PDT
DEBUG: CheckPoint record at (0, 108)
DEBUG: Redo record at (0, 108); Undo record at (0, 0); Shutdown TRUE
DEBUG: NextTransactionId: 514; NextOid: 17199
DEBUG: database system is in production state
Initializing pg_shadow.
Enabling unlimited row width for system tables.
Creating system views.
```

Example 2-16: Output from `initdb` (continued)

```

Loading pg_description.
Setting lastsysoid.
Vacuuming database.
Copying template1 to template0.

Success. You can now start the database server using:

/usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data
or
/usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data -l logfile start

```



You can indicate that PostgreSQL should use a different data directory by specifying the directory location with the `-D` option. This path must be initialized through `initdb`.

When the `initdb` command has completed, it will provide you with information on starting the PostgreSQL server. The first command displayed will start `postmaster` in the foreground. After entering the command as it is shown in Example 2-17, the prompt will be inaccessible until you press CTRL-C on the keyboard to shut down the `postmaster` process.

Example 2-17: Running `postmaster` in the foreground

```

$ /usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data
DEBUG: database system was shut down at 2001-10-12 23:11:00 PST
DEBUG: CheckPoint record at (0, 1522064)
DEBUG: Redo record at (0, 1522064); Undo record at (0, 0); Shutdown TRUE
DEBUG: NextTransactionId: 615; NextOid: 18720
DEBUG: database system is in production state

```

Starting PostgreSQL in the foreground is not normally required. We suggest the use of the second command displayed. The second command will start `postmaster` in the background. It uses `pg_ctl` to start the postmaster service, as shown in Example 2-18.

Example 2-18: Running `postmaster` in the background

```

$ /usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data -l /tmp/pgsql.log start
postmaster successfully started

```

The major difference between the first command and the second command is that the second runs `postmaster` in the background, as well as redirects any debugging information to `/tmp/pgsql.log`. For normal operation, it is generally better to run `postmaster` in the background, with logging enabled.



The `pg_ctl` application can be used to start and stop the PostgreSQL server. See Chapter 9 for more on this command.

Step 9: Configuring the PostgreSQL SysV Script

The SysV script will allow the graceful control of the PostgreSQL database through the use of the SysV runlevel system. The SysV script can be used for starting, stopping, and status-checking of PostgreSQL. It is known to work with most Red Hat based versions of Linux, including Mandrake; however, it should work with other SysV systems (e.g., UnixWare, Solaris, etc.) with little modification. The script is named `linux`. To use it, you will first need to copy the `linux` script to your `init.d` directory. You may require root access to do this.

First, change to the directory where you unpacked the PostgreSQL source. In our case, the path to that directory is `/usr/local/src/postgresql-7.1.3/`. Then, issue a `cp` command to copy the script from `contrib/start-scripts` into the `init.d` directory. Example 2-19 demonstrates how to do this on a Red Hat Linux system.

Example 2-19: Copying the linux script

```
$ cd /usr/local/src/postgresql-7.1.3/  
$ su -c "cp contrib/start-scripts/linux /etc/rc.d/init.d/postgresql"
```

Example 2-19 arbitrarily re-names the new copy to `postgresql`; you may call it whatever you prefer, though it is typically named either `postgresql`, or `postgres`.

You will need to make the script file *executable* after copying it. To do so, use the command shown in Example 2-20.

Example 2-20: Making the linux script executable

```
$ su -c "chmod a+x /etc/rc.d/init.d/postgresql"
```

There are no additional requirements to use the SysV script with Red Hat, if you do not intend on using it to start PostgreSQL automatically (i.e., if you wish to use the script manually). However, if you do wish for the script to startup PostgreSQL automatically when the machine boots up (or changes runlevels), you will need to have the `chkconfig` program installed. If `chkconfig` is installed, you will also need to add the following two lines, including the hash (#) symbol, at the beginning of the `/etc/rc.d/init.d/postgresql` file:

```
# chkconfig: 345 85 15  
# description: PostgreSQL RDBMS
```

These example numbers should work on your system; however, it is good to know what role they perform. The first group of numbers (345) represent which runlevels PostgreSQL should be started at. The example shown would start PostgreSQL at runlevels 3, 4, and 5. The second group of numbers (85) represent the order in which PostgreSQL should be started within that runlevel, relative to other programs. You should probably keep the second number high, to indicate that it should be started later in the runlevel. The third number (15) represents the order in which PostgreSQL should be shutdown. It is a good idea to keep this number low, representing a shutdown order that is inverse from the startup order. As previously mentioned, the script should work on your system with the numbers provided, but you can change them if it is necessary.

Once these two lines have been added to the script, you can use the commands shown in Example 2-21 on Red Hat and Mandrake Linux distributions to start the PostgreSQL database. Be sure to execute these as the *root* user.

Example 2-21: Starting PostgreSQL with the SysV script

```
$ service postgresql start
Starting PostgreSQL: ok
$ service postgresql stop
Stopping PostgreSQL: ok
```



The SysV script logs redirects all PostgreSQL debugging output to `/usr/local/pgsql/data/serverlog`, by default.

Step 10: Creating a Database

Now that the PostgreSQL database system is running, you have the option of using the default database, `template1`. If you create a new database, and you would like all of your consecutive databases to have the same system-wide options, then you should first configure the `template1` database to have those options enabled. For instance, if you plan to use the PL/pgSQL language to program, then you should install the PL/pgSQL language into `template1` before using `createdb`. Then when you use the `createdb` command, the database created will inherit `template1`'s objects, and thus, inherit the PL/pgSQL language. For more information on installing the PL/pgSQL language into a database, refer to Chapter 11, *PL/pgSQL*.

The next step will be to create a new database. This will be a simple test database. We do not recommend using the default `template1` database for testing purposes. As you have not created any users with database-creation rights, you will want to make sure that you are logged in as the *postgres* user when adding a new database. You can also create users that are allowed to add databases, which is

discussed later in Chapter 10, *User and Group Management*. To create a new database named `testdb`, enter the command shown in Example 2-22.

Example 2-22: Creating a database

```
$ createdb testdb
CREATE DATABASE
```

You should receive a message that says `CREATE DATABASE`, indicating that creation of the database was successful. You can now use PostgreSQL's command line interface, *psql*, to access the newly created database. To do so, enter the command shown in Example 2-23.

Example 2-23: Accessing a database with psql

```
$ psql testdb
```

You can now start entering SQL commands (e.g., such as `SELECT`) at the *psql* prompt. If you are unfamiliar with *psql*, please see Chapter 4, *Using SQL with PostgreSQL* for an introduction.

To verify that the database is working correctly, you can issue the command shown in Example 2-24, which should give you a listing of the languages installed in the database.

Example 2-24: Querying a system table

```
testdb=# SELECT * FROM pg_language;
 lanname | lanispl | lanpltrusted | lanplcallfoid | lancompiler
-----+-----+-----+-----+-----
 internal | f       | f             |                | n/a
 C       | f       | f             |                | /bin/cc
 sql     | f       | f             |                | postgres
(3 rows)
```

