dbconfig-common documentation

Sean Finney

Abstract

dbconfig-common is an implementation of the "best practices for database applications" draft, which provides debian packagers with an easy, reliable, and consistent method for managing databases used by debian packages.

Contents

Chapter 1

Introduction

`dbconfig-common` can:

- support MySQL, PostgreSQL and SQLite based applications
- create databases and database users
- access local or remote databases
- upgrade/modify databases when upstream changes database structure
- remove databases and database users
- generate config files in many formats with the database info
- import configs from packages previously managing databases on their own
- prompt users with a set of normalized, pre-translated questions
- handle failures gracefully, with an option to retry.
- do all the hard work automatically
- work for package maintainers with little effort on their part
- work for local admins with little effort on their part
- comply with an agreed upon set of standards for behavior
- do absolutely nothing if it is the whim of the local admin
- perform all operations from within the standard flow of debian package maintenance (no additional skill is required of the local admin)

Chapter 2

Try it out!

If you're interested in trying it out, go ahead and check out 'Using `dbconfig-common` in your packages' on page 5, which will teach you how to get your packages working with `dbconfig-common`.

If you'd like to see some basic examples of `dbconfig-common` in action, check out the sample packages available in `/usr/share/doc/dbconfig-common/examples`. In this directory you'll find debian source directories to build the binary packages, so you can see not only how the packages work when they are installed, but also how they are built. Take a look at the README file in this directory for more information.

Chapter 3

Using dbconfig-common in your packages

3.1 Quick and dirty: what to do

There are three things you will have to do as a package maintainer if you want to use `dbconfig-common`: provide the database code/scripts to setup the data base, source the maintainer script libraries and launch `dbconfig-common`. `dbconfig-common` will take care of everything else, including all debconf related questions, database/database-user creation, upgrade/remove/purge logic, etc. After all, the goal of `dbconfig-common` is to make life easier for both the local admin *and* the package maintainer :)

3.1.1 Update package dependencies

If your package just supports a single database type supported by dbconfig-common, your package needs to depend on the matching `dbconfig-<database-type>` package as well as have the `dbconfig-no-thanks` as an alternative to that. E.g. packages that need a PostgreSQL database must have the following in their Depends field:

```
Depends: dbconfig-pgsql | dbconfig-no-thanks
```

Packages that support multiple database types (see more about that in the 'Packages that support multiple types of databases' on page 7) must alternatively depend on all the matching `dbconfig-<database-type>` packages as well as have `dbconfig-no-thanks` as an alternative to that. E.g. packages that need a PostgreSQL or SQLite3 database must have the following in their Depends field:

```
Depends: dbconfig-pgsql | dbconfig-sqlite3 | dbconfig-no-thanks
```

3.1.2 Putting hooks into the maintainer scripts

In the *config*, *postinst*, *prerm*, and *postrm* scripts for your package, you will need to source the libraries which perform most of the work for you (you do not need to do so in your *preinst* script). If you are not currently using debconf in your package, you will be now, and the debconf libraries need to be sourced first. You will need to use dh_installdebconf or otherwise install your *config* script into your deb file if you're not already doing so. For example, here's what it might look like in a *config* script for an imaginary `foo-mysql` package:

```
#!/bin/sh
# config maintainer script for foo-mysql

# source debconf stuff
. /usr/share/debconf/confmodule
# source dbconfig-common shell library, and call the hook function
if [ -f /usr/share/dbconfig-common/dpkg/config.mysql ]; then
    . /usr/share/dbconfig-common/dpkg/config.mysql
    dbc_go foo-mysql "$@"
fi

# ... rest of your code ...
```

dbc_go is a function defined in every maintainer script hook to execute the appropriate code based on which maintainer script is being run. Note that it is passed two arguments. `foo-mysql`, the name of the package (there's sadly no clean way to figure this out automatically), and *$@* (the arguments which were passed to the maintainer script).

NOTE: you do not need to conditionally test for the existance of the shell library in the *postinst* and *prerm* scripts, but to stay compliant with Policy section 7.2 you do need to do so at least in your *config* and *postrm* scripts.

Note that if your package does not use debconf, you will need to explicitly install the *config* script in your package. The easiest way to do so is to call dh_installdebconf from `debian/rules`.

3.1.3 Supplying the data/code for your database

There are three locations in which you can place code for installing the databases of your package:

- `/usr/share/dbconfig-common/data/PACKAGE/install/DBTYPE`
- `/usr/share/dbconfig-common/data/PACKAGE/install-dbadmin/DBTYPE`
- `/usr/share/dbconfig-common/scripts/PACKAGE/install/DBTYPE`

where *PACKAGE* is the name of the package, *DBTYPE* is the type of data (mysql, pgsql, etc). The full location should be a file containing the proper data.

The first location is for the majority of situations, in which the database can be constructed from it's native language (SQL for MySQL/PostgreSQL, for example). The data will be fed to the underlying database using the credentials of the database user. The second location is like the first location, but will be run using the credentials of the database administrator. *Warning*: use of this second location should only be done when there are excerpts of database code that *must* be run as the database administrator (such as some language constructs in postgresql) and should otherwise be avoided. The third location is for databases that require a more robust solution, in which executable programs (shell/perl/python scripts, or anything else) can be placed.

This code will only be executed on new installs and reconfiguration of failed installs. In the case of SQL databases, in the data directory you would find the simple create and insert statements needed to create tables and populate the database. *You do not need to create the underlying database, only populate it*. The scripts directory contains shell/perl/python/whatever scripts, which are passed the same arguments as *dbc_go*. If you need database connection information (username, password, etc) in your scripts, you can source the `/bin/sh` format package config file, or you can instruct `dbconfig-common` to generate one in your programming language of choice (see the advanced tips section).

If files exist in both data and scripts, they will both be executed in an unspecified order.

That's it! The rest of what needs to be done is handled by `dbconfig-common`, which should keep all the work (and bugs) in one place. Happy packaging! Of course, it's recommended you take a quick look through the rest of the document, just to get an idea of other things that `dbconfig-common` can do for you in case you have special needs.

3.2 Advanced usage.

3.2.1 Generating custom configuration files with database information

Your database application will probably require a username and password in order to function. Every package that uses `dbconfig-common` already has a /bin/sh includable format config file, but it may be more convenient to have something in the native language of the package. For example, packaging a php/MySQL web app would be a lot easier if there were already a file existing with all the information in php includable format.

Using `dbconfig-common`, you can do this with little effort. In your *postinst* script, define the variable *dbc_generate_include* to a value that follows the form *format:location* where *format* is one of the supported output formats of *dbconfig-generate-include* (list them with -h) and location is the absolute location where you want your config file to go. There are also some extra variables *dbc_generate_include_owner*, *dbc_generate_include_perms*, and *dbc_generate_include_args* which do what you would expect them to. *Note: you will be responsible for removing this file in your postrm script*. *Note2: the ownership and permissions will only be used when creating the file, so don't rely on this feature if your packages wants to change existing ownership and/or permissions.* When your scripts are run, this environment variable will be exported to your scripts, as well as a variable *dbc_config_include* which has the same value, but with the leading *format:* stripped away for convenience. *NOTE* if you use this feature, you should also ensure that the generated file is properly removed in the *postrm* script. dbconfig-common can not handle this itself, unfortunately, because it may be possible that it is purged before your package is purged. therefore, you should do the following in your *postrm* script:

```
if [ "$1" = "purge" ]; then
    rm -f yourconfigfile
    if command -v ucf >/dev/null; then
        ucf --purge yourconfigfile
        ucfr --purge yourpackage yourconfigfile
    fi
fi
```

3.2.2 Importing `dbconfig-common` into an existing package

If your package is already part of debian, `dbconfig-common` provides some support to load pre-existing settings from a specified config by setting two variables: *dbc_first_version* and *dbc_load_include*.

dbc_load_include should be specified in the *config* script and be of the format *format:inputfile*. *format* is one of the languages understood by *dbconfig-load-include*, and *inputfile* is either the config file in *format* language, or a script file in *format* language that otherwise determines the values and sets them.

dbc_first_version should be specified in both the *config and postinst* scripts, and should contain the first version in which `dbconfig-common` was introduced. When the package is installed, if it is being upgraded from a version less than this value it will attempt to bootstrap itself with the values.

3.2.3 Database changes in new versions of your package

Occasionally, the upstream authors will modify the underlying databases between versions of their software. For example, in MySQL applications column names may change, move to new tables, or the data itself may need to be modified in newer upstream versions of a package.

In order to cope with this, a second set of file locations exists for providing packagers ways to modify the databases during package upgrades:

- `/usr/share/dbconfig-common/data/PACKAGE/upgrade/DBTYPE/VERSION`
- `/usr/share/dbconfig-common/data/PACKAGE/upgrade-dbadmin/DBTYPE/VERSION`
- `/usr/share/dbconfig-common/scripts/PACKAGE/upgrade/DBTYPE/VERSION`

where *VERSION* is the version at which the upgrade should be applied, and the respective path contains the upgrade code/data. When a package upgrade occurs, all instances of *VERSION* which are newer than the previously installed version will be applied, in order. There is also an automatically included set of safeguards and behavior provided by `dbconfig-common`, so as the packager you shouldn't need to worry about most of the error-handling.

As with installation, scripts will be passed the same cmdline arguments as were passed to dbc_go.

Internally `dbconfig-common` uses dpkg to determine whether an upgrade should be applied. It does this by comparing the lexicographic sort order of the upgrade's code/data *VERSION* filename against the old package's version string as found in its `debian/changelog`. Maintainers can verify in advance, noting an exit status of zero on success, like so:

```
$ OLD_PACKAGE_VERSION=0.4.9-git20240201-1
$ UPGRADE_SCRIPT_FILENAME=0.5.0
$ dpkg --compare-versions $OLD_PACKAGE_VERSION lt $UPGRADE_SCRIPT_FILENAME
$ echo $?
0
```

Note that *$UPGRADE_SCRIPT_FILENAME* in the above example need not match verbatim the actual package version that ships it as found in the new package's `debian/changelog`. This can be useful if the maintainer would prefer to use an abridged filename for the upgrade script, such as `0.5.0` instead of `0.5.0-git20240302-2`.

3.2.4 Packages that support multiple types of databases

Sometimes, a particular package may support multiple database types. This is common with perl or php based web applications, which frequently use some form of database abstraction layer (pear DB for php, the DBD family for perl).

`dbconfig-common` provides support for such applications in a relatively straightforward fashion, allowing the local admin to select which database type to use when configuring a database for a package

To take advantage of this feature, you will want to use the "generic" maintainer script hooks, and additionally hint the debconf *config* script with the types of databases your package supports. For example, the *postinst* script would now look like this:

```
#!/bin/sh
# postinst maintainer script for foo-mysql

# source debconf stuff
. /usr/share/debconf/confmodule
# source dbconfig-common stuff
. /usr/share/dbconfig-common/dpkg/postinst
dbc_go foo-mysql "$@"

# ... rest of your code ...
```

Unfortunately, specifying the proper dependencies on the right `dbconfig-<database-type>` packages, as discussed earlier, is not enough for dbconfig-common to figure out which database types your package supports. Therefore you need to let dbconfig-common know via the *config* script. It needs to contain an additional variable called "dbc_dbtypes", which is a comma-separated list of supported database types:

```
#!/bin/sh
# config maintainer script for foo-mysql

# source debconf stuff
. /usr/share/debconf/confmodule
if [ -f /usr/share/dbconfig-common/dpkg/config ]; then
    # we support mysql and pgsql
    dbc_dbtypes="mysql, pgsql"

    # source dbconfig-common stuff
    . /usr/share/dbconfig-common/dpkg/config
    dbc_go foo-mysql "$@"
fi

# ... rest of your code ...
```

3.2.5 Packages that connect to but should not create databases (read-only frontends)

Some packages provide multiple frontend packages to a single backend package. Furthermore, sometimes these frontend packages are installed on a seperate system from the actual database application, and should not manage the databases on their own.

For example, if the frontend were to be installed on multiple servers (perhaps load balancing or similar), it would not be wise to attempt to install/upgrade the database on each client. Instead, it would be wiser to simply prompt for the information and leave the database management to the single central package.

If the above scenario matches one of your packages, there are a seperate set of maintainer hooks for you to use. For example, `frontend.config` or `frontend.config.mysql`. Using these hooks, `dbconfig-common` will know enough to not take any actions apart from prompting the local administrator for the pertinent information.

3.2.6 Packages that require extra logic during removal

Sometimes, it may be that your install sql/scripts perform operations that aren't automatically undone by package removal. For example, if your package gives extra grants to a user (such as triggers) it's possible that grants will not automatically be revoked, which could cause problems for later installations as well as potential security concerns. For this and any other use you may need it for, you can place files in the following locations for "removal" maintainer code:

- `/usr/share/dbconfig-common/data/PACKAGE/remove/DBTYPE`
- `/usr/share/dbconfig-common/scripts/PACKAGE/remove/DBTYPE`

This works just like the install/upgrade code, only it always runs as the dbadmin. This code is run by default, unless the local admin opts out of deconfiguration assistance (note that this is seperate from database purging, which does not happen by default). Note that if you need to perform template substitution, you should set dbc_sql_substitutions to "yes" in your prerm maintainer script as well.

3.2.7 Hinting defaults and advanced control of configuration/installation

`dbconfig-common` has a set of pre-defined default values for most of the questions with which it prompts the user, most of which are variations on the name of the package. However, as a packager you can override some these values and set defaults that you feel are more appropriate, as well as otherwise modify the behavior of some parts of `dbconfig-common`.

The following table lists the variables you can hint in your *config* script, as well as some other variables you can use to have a finer level of control over `dbconfig-common`. *You must use these variables exactly (and only) where directed in this table.*

dbc_dbuser (used in: *config*) Name to use when connecting to database. (defaults to: package name)

dbc_basepath (used in: *config*) Database storage directory for local (filesystem) based database types. Not applicable for RDBMs like MySQL and postgres. (defaults to: /var/lib/dbconfig-common)

dbc_dbname (used in: *config*) Name of database resource to which to connect. (defaults to: package name)

dbc_dbtypes (used in: *config*) Database types supported by the package, in order of maintainers' preference. (defaults to: empty)

dbc_dbfile_owner (used in: *postinst*) Set the owner:group for the generated database file. This option is only valid for databases like SQLite that use a single file for storage and is not prompted via debconf. (defaults to: root:root)

dbc_dbfile_perms (used in: *postinst*) Set the permissions for the generated database file. This option is only valid for databases like SQLite that use a single file for storage and is not prompted via debconf. (defaults to: 0640)

dbc_generate_include (used in: *postinst*) format:outputfile pair for an extra config to be generated by *dbconfig-generate-include*. (defaults to: empty)

dbc_generate_include_owner (used in: *postinst*) Set the owner:group of include files generated by *dbconfig-generate-include*. (defaults to: empty)

dbc_generate_include_perms (used in: *postinst*) Set the permissions of include files generated by *dbconfig-generate-include*. (defaults to: empty)

dbc_generate_include_args (used in: *postinst*) Arguments passed directly to *dbconfig-generate-include*. (defaults to: empty)

dbc_dgi_on_manual (used in: *postinst*) Control whether config files should be generated by *dbconfig-generate-include* when the admin opts for manual installation. (defaults to: true)

dbc_first_version (used in: *config,postinst*) The first version in which `dbconfig-common` was introduced in the package. (defaults to: empty)

dbc_load_include (used in: *config*) format:includefile pair for a config to be read in by *dbconfig-load-include*. (defaults to: empty)

dbc_load_include_args (used in: *config*) Arguments passed directly to *dbconfig-load-include*. (defaults to: empty)

dbc_pgsql_createdb_encoding (used in: *postinst*) Specifies encoding for created postgres databases. (defaults to: empty/system default)

dbc_mysql_createdb_encoding (used in: *postinst*) Specifies encoding for created MySQL databases. (defaults to: empty/system default)

dbc_sql_substitutions (used in: *postinst*, sometimes *prerm*) If nonempty, specifies that sql files should be piped through a template substitution filter (*dbconfig-generate-include -f template*) before being executed. (defaults to: empty)

dbc_authmethod_user (used in *config*) If set to "ident", dbconfig-common will set the default postgres authentication method for the package's database user to "ident". (defaults to: empty)

dbc_config_allow_backup (used in *config*) If nonempty, dbconfig-common will allow its state machine to backup past the beginning, such that packages allowing backup in there own config script to work transparently.

dbc_prio_low / dbc_prio_medium / dbc_prio_high / dbc_prio_critical (used everywhere) If nonempty, dbconfig-common will use the values of these variables to set the priority of its debconf questions. You can use this if you think that the default levels of dbconfig-common are not well defined for your package. Be comforted thou that the priority will be raised automatically on errors and that all error handling allows the sysadmin to be asked the questions again.

dbc_authplugin (used in *config*) This option defines which MySQL authentication plugin should be set to default whenever dbconfig-common creates a user in the database on behalf of the package using it. Choices are:

- default: use the default auth plugin for installed MySQL server.
- mysql_native_password: no MySQL authentication plugin is used.
- sha256_password: more secure password encryption than native.
- caching_sha2_password: in-memory authentication cache.

dbc_authplugin should be set in .config file, right before calling dbc_go function. It can also be configured by user during package installation, or during dpkg-reconfigure, if debconf questions priority is set to low.

Be careful when changing this. There are some differences in between MySQL and MariaDB and some options might not be available on one or another. As up to MariaDB 10.4, its default authentication plugin is “mysql_native_password” while, for MySQL 8.0, the new default authentication plugin is the “caching_sha2_password”.

3.2.8 Debugging problems with `dbconfig-common`

In the event that your package is having trouble working with `dbconfig-common`, the first thing you should try is to export and set the shell variable *dbc_debug* to a nonempty value before installing your package. This will provide a slightly larger amount of information about what’s going on.

In the event that this does not provide enough information, the next thing to do will provide much, much, more information; enough that you will probably want to redirect stderr into a temporary output file. In the file `/usr/share/dbconfig-common/dpkg/common`, uncomment the *set -x* line near the top of the file. This will show you all the shell commands and logic as they are executed. If you have a good idea of where the problem is occurring, you can also insert your own *set -x* lines elsewhere followed by *set +x* lines to reduce the amount of input.

Chapter 4

More Information (and how to help)

Currently, there's a fair amount of work left to be done:

- more translators/translations are needed for the templates
- developers are needed to volunteer their packages with `dbconfig-common`
- volunteers are needed to test the new packages
- support for other database formats would be nice
- more scheduled features are listed in `/usr/share/doc/dbconfig-common/TODO`