COMP3311 18s1 Database Systems

Prac Exercise 02
Setting up your PostgreSQL server

Aims

This exercise aims to get you to:

- set up your "virtual host"
- install your PostgreSQL database server on the virtual host
- create, populate and examine a very small database

You ought to get it done by end of Week3 since you'll need it to start working on Assignment 1.

You can also install PostgreSQL on your home machine if you want to work there, but you'll need to work out how to do that yourself. There are plenty of online resources describing how to do this for different operating systems (type "install postgresql" at Google). It probably doesn't matter too much which version you install, since we'll be using a subset of SQL and PLpgSQL that hasn't changed for a l-o-n-g time. Of course, you should *always* test your work on the CSE machines before you submit, since that's where we'll be testing them.

Background

The practical work for the assignments will be carried out on the CSE server called `grieg`, using a "virtual host". On this virtual host, you can run your own PostgreSQL database server (for which you are effectively the database administrator).

Since everyone in the course can run a PostgreSQL server on `grieg`, the basic task of the virtual host provides a private IP address on which you can run your own server without interfering with other people's servers. Everyone who's enrolled in COMP3311 should have the ability to set up a virtual host. If you cannot log in to `grieg` or run the `priv srvr` command on `grieg`, let us know.

In the examples below, we have used the `$` sign to represent the prompt from the command interpreter (shell). In fact, the prompt may look quite different on your machine (e.g., it may contain the name of the machine you're using, or your username, or the current directory name). All of the things that the computer types are in `this font`. The commands that **you** are supposed to type are in **`this bold font`**. Some commands use *`YOU`* as a placeholder for your CSE username; do not type the three characters "YOU".

Exercises

Stage 1: Creating a Virtual Host

Log in to `grieg.cse.unsw.edu.au`. If you're not logged into `grieg` nothing that follows will work.

You can log into `grieg` from a command-line (shell) window on any CSE lab workstation via the command:

```
$ ssh grieg
```

If you're doing this exercise from home, you can use any `ssh` client, but you'll need to refer to `grieg` via its fully-qualified name:

```
$ ssh CSEUsername@grieg.cse.unsw.edu.au
```

You can check whether you're actually logged in to `grieg` by using the command:

```
$ hostname
```

Once you're logged into `grieg`, run the command:

```
$ priv srvr
```

This will do two things:

- create a new directory (`/srvr/`*`YOU`*`/`) to hold your server files
- create a unique and permanent IP address for your virtual host

After you have run the `priv srvr` command, you can find out your host's IP address via the command:

```
$ cat /proc/self/ipv4root
```

(If the above prints `0.0.0.0`, it means that you are not currently running a virtual host).

While it might be interesting to know the IP address, it's not essential. The above command is probably more useful as a way of checking that you have actually "started" your virtual server.

Once you've set up your environment, you can check your virtual host's IP address more easily via the command:

```
   $ ~cs3311/bin/my_ip
```

or, when your environment is completely set up, by simply:

```
  $ my_ip
```

Since you'll be using several commands from the `~cs3311/bin/` directory, the `env` script that's installed later in this exercise will make your life easier by adding this directory into your command path. You should ensure that you invoke the `env` script whenever you're using `grieg` (see below for more info).

As noted above, the other thing that the `priv srvr` command will do the first time you run it, is create a directory called

```
/srvr/YOU/
```

This directory is initially empty, but will eventually contain all of the data files for your PostgreSQL server. You can also place other COMP3311-related files under this directory. Make sure, however, that any directories containing assignment work are not accessible to other people.

Each time you want to use your PostgreSQL server, you will need to log in to `grieg` and run the `priv srvr` command.

> **Mini-Exercise**: log out from `grieg`, then log back in and use the `my_ip` command to check the IP address. If the system tells you that it can't find the `my_ip` command or it tells you that the IP address is `0.0.0.0`, then fix the "problem" (i.e. ensure that you're using the full path of the `my_ip` command and that it displays a non-zero IP address).

Stage 2: Setting up your PostgreSQL Server

Once the `/srvr/YOU/` directory exists and after you have run the `priv srvr` command, you are ready to set up your PostgreSQL server. Do this by running the command:

```
$ ~cs3311/bin/pginit
```

This command will create a subdirectory called `pgsql` in your `/srvr/YOU/` directory, and also place a file called `env` in `/srvr/YOU/`.

> The `pginit` script checks for every error that I could think of, but I'm sure you'll find some others, so let me know if you get any error messages when you run `pginit`. The script *does* produce some messages from PostgreSQL when it's creating things, but these are *not* errors. Even the warnings are nothing to worry about (for this course).

The output from `pginit` should look something like:

```
$ /srvr/cs3311psql/bin/pginit
Installing environment setup script ...
successful.

Each time you log you need to set your enviornment by running:
   source /srvr/YOU/env
Note that you *must* use the full path name for this file.

Installing PostgreSQL data directories ...
The files belonging to this database system will be owned by user "YOU".
This user must also own the server process.
...
a whole lot more stuff which you can ignore
as long as it doesn't have any ERRORs
...
server stopped
PostgreSQL installed ok.
To start the server:
* log in to grieg; run 'priv srvr'
* source the '/srvr/YOU/env' file
* run the command 'pgs start'
To stop the server:
* run the command 'pgs stop'
```

where, obviously, your login name will appear in place of *YOU*.

You can ignore any WARNINGs which may appear in the `pginit` output. They have no effect on the usefulness of your PostgreSQL server. In general, however, you should not ignore WARNING messages and should never ignore ERROR messages from PostgreSQL.

You should only run the `pginit` command once (unless you need to completely reinstall your PostgreSQL server from scratch).

> One place where PostgreSQL is less space efficient than it might be is in the size of its transaction logs. These logs live in the directory `pgsql/pg_xlog` and are essential for the functioning of your PostgreSQL server. If you remove any files from this directory, you will

render your server inoperable. Similarly, manually changing the files under `pgsql/base` and its subdirectories will probably break your PostgreSQL server.

If you mess up your PostgreSQL server badly enough, it will need to be re-installed. If such a thing happens, all of your databases are useless and all of the data in them is irretrievable. You will need to completely remove the `/srvr/YOU/pgsql` directory and re-install using `pginit`.

If you need to remove the `pgsql` directory, then all of your databases and any data in them are gone forever. This is not a problem if you set up your databases by loading new views, functions, data, etc. from a file, but if you type `create` commands directly into the database, then the created objects will be lost. The best way to avoid such catastrophic loss of data is to type your SQL `create` statements into a file and load them into the database from there. Alternatively, you'd need to do regular back-ups of your databases using the `pg_dump` command.

The `env` file that `pginit` places in your `/srvr/YOU/` directory contains a bunch of environment settings that need to be active before the servers will work. Since these environment settings need to affect your `grieg` login shell, you must **source** the `env` file, not execute it. You could do this, once you're logged in to `grieg`, via:

```
$ source /srvr/YOU/env
```

Remembering to do this each time you log in to `grieg` is slightly annoying. You can simplify things so that the environment gets set up automatically when you login to `grieg`. To do this, add the following code at the end of your `.bashrc` or `.bash_profile` file:

```
if [ `hostname` = "grieg" ]
then
    priv srvr
    source /srvr/YOU/env
fi
```

Note that the quotes around `hostname` are back-quotes. Note also that the spaces (except at the start of each line) are critically important for making this run correctly. If you don't already have a `.bashrc` file, and don't know what one is, you might want to skip the above step. If you don't do it, it simply means that each time you log in to `grieg`, you'll need to run the commands:

```
$ priv srvr
$ source /srvr/YOU/env
```

before you do anything with PostgreSQL.

Stage 3: Using your PostgreSQL Server

When you want to do some work with PostgreSQL: login to Grieg, start your server, do your work, and then stop the server before logging off Grieg.

Do not leave your PostgreSQL server running while you are not using it.

The command for controlling your PostgreSQL server is:

```
$ ~/cs3311/bin/pgs
```

Once you've set up your environment properly, you should be able to invoke this command simply by typing `pgs`.

Each time you want to use your PostgreSQL server, you'll need to do the following:

```
$ ssh grieg                     log in to grieg
... starts a new login session on Grieg ...
$ priv srvr                     start your virtual host
$ source /srvr/YOU/env          set up your environment
$ pgs start                     start the PostgreSQL server
```

Remember to do all of the above each time you login to the CSE machines to do some work with PostgreSQL.

If you ever get an error message like this:

```
$ pgs start
-bash: pgs: command not found
```

then you probably haven't set up your shell `PATH` properly.

If you see the above, you need to `source` your `env` file.

You can check whether your server is running via the command:

```
$ pgs status
```

If you do have a server running, this command will give you output from the Unix `ps` command showing the PostgreSQL processes that you currently have running. There should be one process that looks like:

```
bin/postgres
```

and a couple of PostgreSQL `writer` processes. If this does not show at least one `postgres` process, then your PostgreSQL server is not running.

You can stop your server via the command:

```
$ pgs stop
```

Try checking, stopping, and starting the server a few times.

Things occasionally go wrong, and knowing how to deal with them will save you lots of time. There's a discussion of common problems at the end of this document; make sure that you read and understand it.

Once your PostgreSQL server is running, you can access your PostgreSQL databases via the `psql` command. You normally invoke this command by specifying the name of a database, e.g.

```
$ psql MyDatabase
```

If you type `psql` command without any arguments, it assumes that you are trying to access a database with the same name as your login name. Since you probably won't have created such a database, you're likely to get a message like:

```
psql: FATAL:  database "YOU" does not exist
```

You will get a message like this any time that you try to access a database that does not exist.

If you're not sure what databases you have created, `psql` can tell you via the `-l` option (that's lower-case 'L' not the digit '1' (one)), e.g.

```
$ psql -l
```

If you run this command now, you ought to see output that looks like:

```
                              List of databases
   Name    | Owner | Encoding |  Collate   |   Ctype    | Access privileges
-----------+-------+----------+------------+------------+-------------------
 postgres  | YOU   | LATIN1   | C          | en_AU      |
 template0 | YOU   | LATIN1   | C          | en_AU      | =c/YOU           +
           |       |          |            |            | YOU=CTc/YOU
 template1 | YOU   | UTF8     | en_US.utf8 | en_US.utf8 |
(3 rows)
```

Note that PostgreSQL commands like `psql` and `createdb` are a lot noisier than normal Linux commands. In particular, they all seem to be printing `SET` when they run; you can ignore this. Similarly, if you see output like `INSERT 0 1`, you can ignore that as well.

These three databases are created for use by the PostgreSQL server; you should not modify them. At this stage, you don't need to worry about the contents of the other columns in the output. As long as you see at least three databases when you run the `psql -l` command, it means that your PostgreSQL server is up and running ok.

The way we have set up the PostgreSQL servers on `grieg`, each student is the administrator for their own server. This means that you can create as many databases as you like (until you run out of disk quota), and make any other changes that you want to the server configuration.

From within `psql`, the fact that you are an administrator is indicated by a prompt that looks like

```
dbName=#
```

rather than the prompt for database users

```
dbName=>
```

which you may have seen in textbooks or notes.

Note that you can only access databases while you're logged into your virtual host on `grieg`. In other words, you must run the `psql` command under your virtual host on `grieg`.

Note that the **only** commands that you should run on `grieg` are the `pgs` command (to start and stop the server), the `psql` command to start an interactive session with a database, and the other PostgreSQL clients such as `createdb`. Do not run other processes such as web browsers, drawing programs or editors on `grieg`. If you do, `grieg` will eventually be overwhelmed and you'll effectively be a contributor to a Denial of Service attack.

If you need to edit files while you're using your PostgreSQL server, run another terminal window on the local machine (not Grieg), and do the editing there. You don't need to login to Grieg to access files in the `/srvr/YOU/` directory, so you can run your terminal session on the local machine and still access your files on Grieg.

All of the PostgreSQL client applications are documented in the PostgreSQL manual. While there are quite a few of them, `psql` will be the one that you will mostly use.

Mini-Exercise: a quick way to check whether your PostgreSQL server is running is to try the command:

```
$ psql -l
```

Try this command now.

If you get a response like:

```
psql: command not found
```

then you haven't set up your environment properly; `source` the `env` file.

If you get a response like:

```
psql: could not connect to server: No such file or directory
        Is the server running locally and accepting
        connections on Unix domain socket "....s.PGSQL.5432"?
```

then the server isn't running.

If you get a list of databases, like the example above, then this means your server is running ok and ready for use.

Cleaning up

After you've finished a session with PostgreSQL, it's essential that you shut your PostgreSql server down (to prevent overloading `grieg`). You can do this via the command:

```
$ pgs stop
```

which must be run on your virtual host (i.e. after you've logged into `grieg` and run `priv srvr`).

PostgreSQL generates log files that can potentially grow quite large. If you start your server using `pgs`, the log file is called

```
/srvr/YOU/pgsql/Log
```

It would be worth checking every so often to see how large it has become. To clean up the log, simply stop the server and remove the file. Note: if you remove the logfile while the server is running, you may not remove it at all; it's link in the filesystem will be gone, but the disk space will continue to be used and grow until the server stops.

Mini-Exercise: Try starting and stopping the server a few times, and running `psql` both when the server is running and when it's not, just to see the kinds of messages you'll get.

Summary

A typical session with your virtual host and your PostgreSQL server would be something like:

```
... on any CSE workstation ...
$ ssh grieg
... grieg login stuff ...
... the following are all on grieg ...
$ priv srvr
$ source /srvr/YOU/env
$ pgs start
$ psql MyDatabase
... use another xterm for editting ...
$ pgs stop
$ logout
... back to your original workstation ...
```

Exercise #1: Making a database

Once the PostgreSQL server is running, try creating a database by running the command:

```
$ createdb mydb
```

which will create the database, or give an error message if it can't create it for some reason. (A typical reason for failure would be that your PostgreSQL server is not running.)

Now use the `psql -l` command to check that the new database exists.

You can access the database by running the command:

```
$ psql mydb
```

which should give you a message like

```
SET
psql (9.4.6)
Type "help" for help.

mydb=#
```

Note that `psql` lets you execute two kinds of commands: SQL queries and updates, and `psql` "meta"-commands. The `psql` "meta"-commands allow you to examine the database schema, and control various aspects of `psql` itself, such as where it writes its output and how it formats tables.

Getting back to the `psql` session that you just started, the `mydb` database is empty, so there's not much you can do with it. The `\d` (describe) command allows you to check what's in the database. If you type it now, you get the unsurprising response

```
mydb=# \d
No relations found.
```

About the only useful thing you can do at the moment is to quit from `psql` via the `\q` command.

```
mydb=# \q
$ ... now waiting for you to type Linux commands ...
```

Note: it is common to forget which prompt you're looking at and sometimes type Linux commands to `psql` or to type SQL queries to the Linux command interpreter. It usually becomes apparent fairly quickly what you've done wrong, but can initially be confusing when you think that the command/query is not behaving as it should. Here are examples of making the above two mistakes:

```
$ ... Linux command interpreter ...
$ select * from table;
-bash: syntax error near unexpected token `from'
$ psql mydb
... change context to PostgreSQL ...
mydb=# ls -l
mydb-# ... PostgreSQL waits for you to complete what it thinks is an SQL query ...
mydb-# ;    ... because semi-colon finishes an SQL query ...
ERROR:  syntax error at or near "ls" at character 1
LINE 1: ls -l
        ^
mydb=# \q
$ ... back to Linux command interpreter ...
```

Exercise #2: Populating a database

Once the `mydb` database exists, the following command will create the schemas (tables) and populate them with tuples:

```
$ psql mydb -f /home/cs3311/web/18s1/pracs/02/mydb.sql
```

Note that this command produces quite a bit of output, telling you what changes it's making to the database. The output should look like:

```
SET
CREATE TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
CREATE TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
CREATE TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
CREATE TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
```

The lines containing `CREATE TABLE` are, obviously, related to PostgreSQL creating new database tables (there are four of them). The lines containing `INSERT` are related to PostgreSQL adding new tuples into those tables.

Clearly, if we were adding hundreds of tuples to the tables, the output would be very long. You can get PostgreSQL to stop giving you the `INSERT` messages by using the `-q` option to the `psql` command.

PostgreSQL's output can be verbose during database loading. If you want to ignore everything *except* error messages, you could use a command like:

```
$ ( psql mydb -f /home/cs3311/web/18s1/pracs/02/mydb.sql 2>&1 ) | grep ERROR
```

If you don't understand the fine details of the above, take a look at the documentation for the Unix shell.

The `-f` option to `psql` tells it to read its input from a file, rather than from standard input (normally, the keyboard). If you look in the `mydb.sql` file, you'll find a mix of table (relation) definitions and statements to insert tuples into the database. We don't expect you to understand the contents of the file at this stage.

If you try to run the above command again, you will generate a heap of error messages, because you're trying to insert the same collection of tables and tuples into the database, when they've already been inserted.

Note that the tables and tuples are now permanently stored on disk. If you switch your PostgreSQL server off, when you restart it the contents of the `mydb` database will be available, in whatever state you left them from the last time you used the database.

Exercise #3: Examining a database

One simple way to manipulate PostgreSQL databases is to use the `psql` command (which is a shell like the `sqlite3` command in the first prac exercise). A useful way to start exploring a database is to find out what tables it has. We saw before that you can do this with the \d (describe) command. Let's try that on the newly-populated `mydb` database.

```
mydb=# \d
         List of relations
 Schema |    Name    | Type  | Owner
--------+------------+-------+-------
 public | courses    | table | YOU
 public | enrolment  | table | YOU
 public | staff      | table | YOU
 public | students   | table | YOU
(4 rows)
```

You can ignore the `Schema` column for the time being. The `Name` column tells you the names of all tables (relations) in the current database instance. The `Type` column is obvious, and, you may think, unnecessary. It's there because `\d` will list all objects in the database, not just tables; it just happens that there are only tables in this simple database. The `Owner` should be your username, for all tables.

One thing to notice is that the table names are all in lower-case, whereas in the `mydb.sql` file, they had an initial upper-case letter. The SQL standard says that case does not matter in identifiers and so `Staff` and `staff` and `STAFF` and even `StAfF` are all equivalent. To deal with this, PostgreSQL simply maps *identifiers* into all lower case internally. You can still use `Staff` when you're typing in SQL commands; it will be mapped automatically before use.

There are, however, advantages to using all lower case whenever you're dealing with `psql`. For one thing, it means that you don't have to keep looking for the shift-key. More importantly, `psql` provides table name and field name completion (you type an initial part of a table name, then type the TAB key, and `psql` completes the name for you if it has sufficient context to determine this unambiguously), but it only works when you type everything in lower case. The `psql` interface has a number of other features (e.g. history, command line editing) that make it very nice to use.

If you want to find out more details about an individual table, you can use:

```
mydb=# \d Staff
             Table "public.staff"
  Column  |         Type          | Modifiers
----------+-----------------------+-----------
 userid   | character varying(10) | not null
 name     | character varying(30) |
 position | character varying(20) |
 phone    | integer               |
Indexes:
    "staff_pkey" PRIMARY KEY, btree (userid)
Referenced by:
    TABLE "courses" CONSTRAINT "courses_lecturer_fkey" FOREIGN KEY (lecturer) REFERENCES staff(userid)
```

As you can see, the complete name of the table is `public.staff`, which includes the schema name. PostgreSQL has the notion of a "current schema" (which is the schema called `public`, by default), and you can abbreviate table names by omitting the current schema name, which is what we normally do. The types of each column look slightly different to what's in the `mydb.sql` file; these are just PostgreSQL's internal names for the standard SQL types in the schema file. You can also see that the `userid` field is not allowed to be null; this is because it's the primary key (as you can see from the index description) and primary keys may not contain null values. The index description also tells you that PostgreSQL has built a B-tree index on the `userid` field.

The final line in the output tells you that one of the other tables in the database (`Courses`) has a foreign key that refers to the primary key of the `Staff` table, which you can easily see by looking at the `mydb.sql` file. This is slightly useful for a small database, but becomes extremely useful for larger databases with many tables.

The next thing we want to find out is what data is actually contained in the tables. This requires us to use the SQL query language, which you may not know yet, so we'll briefly explain the SQL statements that we're using, as we do them.

We could find out all the details of staff members as follows:

```
mydb=# select * from Staff;
  userid  |     name      |     position     | phone
----------+---------------+------------------+-------
 jingling | Jingling Xue  | Professor        | 54889
 jas      | John Shepherd | Senior Lecturer  | 56494
 andrewt  | Andrew Taylor | Senior Lecturer  | 55525
(3 rows)
```

The SQL statement says, more or less, "tell me everything (*) about the contents of the `Staff` table". Each row in the output below the heading represents a tuple in the table.

Note that the SQL statement ends with a semi-colon. The meta-commands that we've seen previously didn't require this, but SQL statements can be quite large, and so, to allow you to type them over several lines, the system requires you to type a semi-colon to mark the end of the SQL statement.

If you forget to put a semi-colon, the prompt changes subtly:

```
mydb=# select * from Staff
mydb-#
```

This is PostgreSQL's way of telling you that you're in the middle of an SQL statement and that you'll eventually need to type a semi-colon. If you then simply type a semi-colon to the second prompt, the SQL statement will execute as above.

Mini-Exercise: find out the contents of the other tables.

Here are some other SQL statements for you to try out. You don't need to understand their structure yet, but they'll give you an idea of the kind of capabilities that the SQL language offers.

- Which students are studying for a CS degree (3978)?
  ```
  select * from Students where degree=3978;
  ```
- How many students are studying for a CS degree?
  ```
  select count(*) from Students where degree=3978;
  ```
- Who are the professors?
  ```
  select * from Staff where position ilike '%professor%';
  ```
- How many students are enrolled in each course?
  ```
  select course,count(*) from Enrolment group by course;
  ```
- Which courses is Andrew Taylor teaching?
  ```
  select c.code, c.title
  from   Courses c, Staff s
  where  s.name='Andrew Taylor' and c.lecturer=s.userid;
  ```

The last query is laid out as we normally lay out more complex SQL statements: with a keyword starting each line, and each clause of the SQL statement starting on a separate line.

Try experimenting with variations of the above queries.

Sorting out Problems

It is very difficult to diagnose problems with software over email, unless you give sufficient details about the problem. An email that's as vague as "My PostgreSQL server isn't working. What should I do?", is basically useless. Any email about problems with software should contain details of

- what you were attempting to do
- precisely what commands you used
- exactly what output you got

One way to achieve this is to copy-and-paste the last few commands and responses into your email.

Alternatively, you should come to a consultation where we can work through the problem on a workstation (which is usually very quick).

Can't shut server down?

When you use `pgs stop` to shut down your PostgreSQL server, you'll observe something like:

```
$ pgs stop
Using server in /srvr/YOU/pgsql
waiting for server to shut down....
```

Dots will keep coming until the server is finally shut down, at which point you will see:

```
$ pgs stop
Using server in /srvr/YOU/pgsql
waiting for server to shut down........ done
server stopped
```

Sometimes, you'll end up waiting for a long time and the server still doesn't shut down. This is typically because you have an `psql` session running in some other window (the PostgreSQL server won't shut down until all clients have disconnected from the server). The way to fix this is to find the `psql` session and end it. If you can find the window where it's running, simply use `\q` to quit from `psql`. If you can't find the window, or it's running from a different machine (e.g. you're in the lab and find that you left a `psql` running at home), then use `ps` to find the process id of the `psql` session and stop it using the Linux `kill` command.

Can't restart server?

Occasionally, you'll find that your PostgreSQL server was not shut down cleanly the last time you used it and you cannot re-start it next time you try to use it. We'll discuss how to solve that here ...

The typical symptoms of this problem are that you log in to `grieg`, set up your environment, try to start your PostgreSQL server and you get the message:

```
PGDATA=/srvr/YOU/pgsql
pg_ctl: another server may be running; trying to start server anyway
server starting   ... this really means "I'm trying to start the server"
!!!
The PostgreSQL server may not have started correctly.
First try the 'psql -l' command to see if it is actually working.
If it's not, then check at the end of the log file for more details.
The log file is called: /srvr/YOU/pgsql/Log
```

If you actually go and check the log file, you'll probably find, right at the end, something like:

```
$ tail -2 /srvr/YOU/logfile
FATAL:  lock file "postmaster.pid" already exists
HINT:  Is another postmaster (PID NNNN) running in data directory "/srvr/YOU/pgsql"?
```

where *NNNN* is a number.

There are two possible causes for this: the server is already running, or the server did not terminate properly after the last time you used it. You can check whether the server is currently running by the command `psql -l`. If that gives you a list of your databases, then you simply forgot to shut the server down last time you used it and it's ready for you to use again. If `psql -l` tells you that there's no server running, then you'll need to do some cleaning up before you can restart the server ...

When the PostgreSQL server is run, it keeps a record of the Unix process that it's running as in a file called:

```
/srvr/YOU/pgsql/postmaster.pid
```

Normally when your PostgreSQL server process terminates (e.g. via `pgs stop`), this file will be removed. If your PostgreSQL server stops, and this file persists, then `pgs` becomes confused and thinks that there is still a PostgreSQL server running even though there isn't.

The first step in cleaning up is to remove this file:

```
$ rm /srvr/YOU/pgsql/postmaster.pid
```

You should also clean up the socket files used by the PostgreSQL server. You can do this via the command:

```
$ rm /srvr/YOU/pgsql/.s*
```

Once you've cleaned all of this up, then the `pgs` command ought to allow you to start your PostgreSQL server ok.