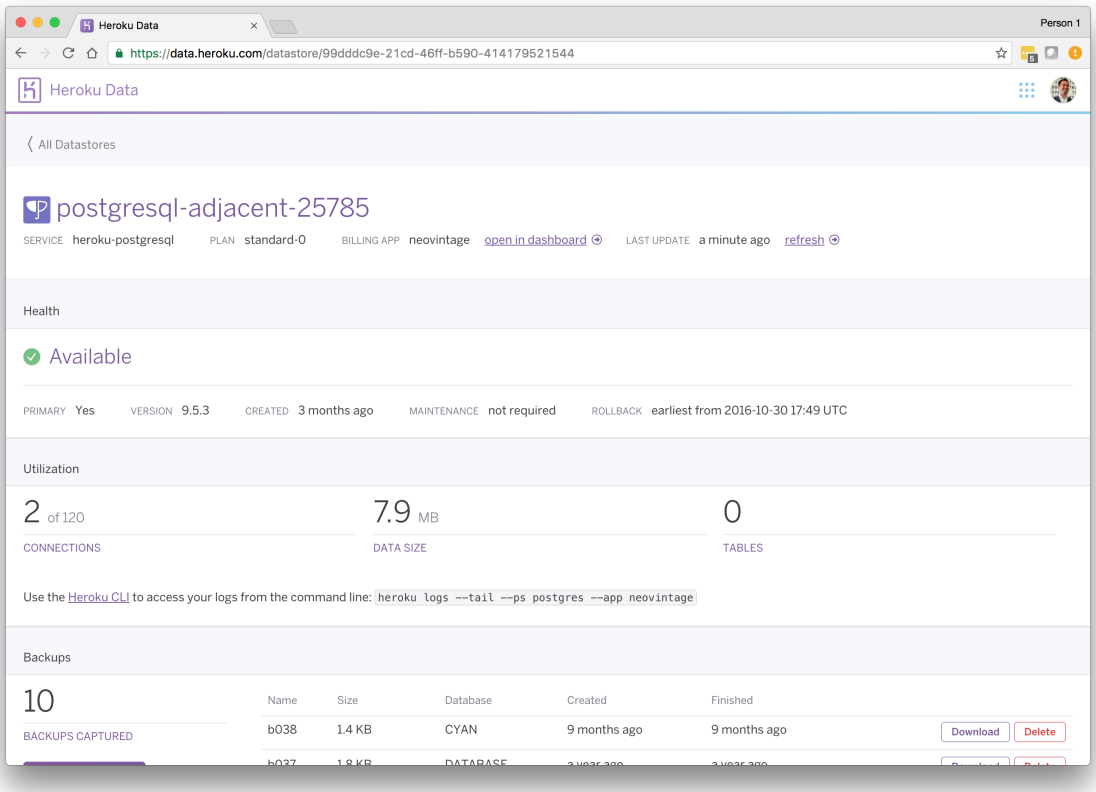
Heroku Postgres

Last updated 04 February 2020

Table of Contents

Heroku Postgres (https://elements.heroku.com/addons/heroku-postgresql) is a managed SQL database service provided directly by Heroku. You can access a Heroku Postgres database from any language with a PostgreSQL driver, including all languages officially supported by Heroku (https://www.heroku.com/languages).



In addition to a variety of management commands available via the Heroku CLI, Heroku Postgres provides a web dashboard (https://data.heroku.com), the ability to share queries with dataclips (https://devcenter.heroku.com/articles/dataclips), and several other helpful features.

Provisioning Heroku Postgres

Before you provision Heroku Postgres, confirm that it isn't *already* provisioned for your app (Heroku automatically provisions Postgres for apps that include certain libraries, such as the `pg` Ruby gem).

Use the `heroku addons` command to determine whether your app already has Heroku Postgres provisioned:

```
$ heroku addons
Add-on                                                  Plan        Price  State
──────────────────────────────────────────────────────  ──────────  ─────  ───────
heroku-postgresql (postgresql-concave-52656)            hobby-dev   free   created
```

If `heroku-postgresql` doesn't appear in your app's list of add-ons, you can provision it with the following CLI command:

```
$ heroku addons:create heroku-postgresql:<PLAN_NAME>
```

For example, to provision a hobby-dev (https://devcenter.heroku.com/articles/heroku-postgres-plans#hobby-tier) plan database:

```
$ heroku addons:create heroku-postgresql:hobby-dev
Creating heroku-postgresql:hobby-dev on ⬢ sushi... free
Database has been created and is available
 ! This database is empty. If upgrading, you can transfer
 ! data from another database with pg:copy
Created postgresql-concave-52656 as DATABASE_URL
```

You can specify the version of Postgres you want to provision by including the `--version` flag in your provisioning command:

```
$ heroku addons:create heroku-postgresql:<PLAN_NAME> --version=9.5
```

Learn more about PostgreSQL version support.

Depending on the plan you choose, your database can take up to 5 minutes to become available. You can track its status with the `heroku pg:wait` command, which blocks until your database is ready to use.

As part of the provisioning process, a `DATABASE_URL` config var is added to your app's configuration. This contains the URL your app uses to access the database. If your app already has a Heroku Postgres database and you've just provisioned *another* one, this config var's name instead has the format `HEROKU_POSTGRESQL_<COLOR>_URL` (for example, `HEROKU_POSTGRESQL_YELLOW_URL`).

You can confirm the names and values of your app's config vars with the `heroku config` command.

> The value of your app's `DATABASE_URL` config var might change at any time. You should not rely on this value either inside or outside your Heroku app.

At this point, an empty PostgreSQL database is provisioned. To populate it with data from an existing data source, see the import instructions (https://devcenter.heroku.com/articles/heroku-postgres-import-export#import) or follow the language-specific instructions in this article to connect from your application.

Understanding Heroku Postgres plans

Heroku Postgres offers a variety of plans, spread across different tiers of service: hobby, standard, premium, and enterprise. For more information on what each plan provides, see Choosing the Right Heroku Postgres Plan (https://devcenter.heroku.com/articles/heroku-postgres-plans).

Pricing information for Heroku Postgres plans is available on the Heroku Postgres add-on page (https://elements.heroku.com/addons/heroku-postgresql).

If your app's requirements eventually outgrow the resources provided by the initial plan you select, you can easily upgrade your database (https://devcenter.heroku.com/articles/upgrading-heroku-postgres-databases#upgrading-with-pg-copy).

Designating a primary database

Heroku apps use the `DATABASE_URL` config var to designate the URL of an app's primary database. If your app only has one database, its URL is automatically assigned to this config var.

For apps with multiple databases, you can set the primary database with the `heroku pg:promote` command:

```
$ heroku pg:promote HEROKU_POSTGRESQL_YELLOW
Promoting HEROKU_POSTGRESQL_YELLOW_URL to DATABASE_URL... done
```

You can specify the "color" identifier or add-on name associated with the database you want to promote.

Sharing Heroku Postgres between applications

You can share a single Heroku Postgres database between multiple apps with the `heroku addons:attach` command:

```
$ heroku addons:attach my-originating-app::DATABASE --app sushi
Attaching postgresql-addon-name to sushi... done
Setting HEROKU_POSTGRESQL_BRONZE vars and restarting sushi... done, v11
```

The attached database's URL is assigned to a config var with the name format `HEROKU_POSTGRESQL_[COLOR]_URL`. In the above example, the config var's name is `HEROKU_POSTGRESQL_BRONZE_URL`.

A shared database is not necessarily the primary database for any given app that it's shared with. You promote a shared database with the same command that you use for any other database.

You can stop sharing your Heroku Postgres instance with another app with the `heroku addons:detach` command:

```
$ heroku addons:detach HEROKU_POSTGRESQL_BRONZE --app sushi
Detaching HEROKU_POSTGRESQL_BRONZE to postgresql-addon-name from sushi... done
Unsetting HEROKU_POSTGRESQL_BRONZE config vars and restarting sushi... done, v11
```

Version support

The PostgreSQL project releases new major versions on a yearly basis. Each major version is supported by Heroku Postgres shortly after its release.

Heroku Postgres supports at least 3 major versions at a given time. Currently supported versions are:

- 12 (default)
- 11
- 10
- 9.6
- 9.5
- 9.4 - deprecating

Users are required to upgrade roughly once every three years. However, you can upgrade your database at any point to gain the benefits of the latest version.

Migration of deprecated databases

The PostgreSQL project stops supporting a major version five years after its initial release (https://www.postgresql.org/support/versioning/). Heroku Postgres deprecates these versions to ensure no databases run on an unsupported major version of PostgreSQL.

- One year before a version's end of life (EOL), Heroku prevents provisioning a new database on the deprecated version (forks and followers of existing databases are still allowed).
- Six (6) months before EOL, customers receive a notification that they must upgrade and directions on how to do so on your own schedule.
- Three (3) months before EOL, we schedule forced upgrade maintenances for any remaining databases.

Heroku **highly recommends** that you perform the version upgrade (https://devcenter.heroku.com/articles/upgrading-heroku-postgres-databases) or update (https://devcenter.heroku.com/articles/updating-heroku-postgres-databases) prior to support ending, so that you can test compatibility, plan for unforeseen issues, and migrate your database on your own schedule.

Legacy Infrastructure

Heroku also occasionally deprecates old versions of its infrastructure. This typically occurs for one of the following reasons:

- The operating system running beneath the database will soon no longer receive security updates.
- Support for the operating system is no longer practical due to its age (if required packages and patches are no longer available or difficult to support).
- The server instances are significantly different from Heroku's current infrastructure and are impractical to support.

To see if your database is running on legacy infrastructure, use `pg:info`:

```
$ heroku pg:info

=== HEROKU_POSTGRESQL_MAROON_URL (DATABASE_URL)
Plan:           Ronin
Status:         Available
Data Size:      26.1 MB
Tables:         5
PG Version:     9.5.3
Connections:    2
Fork/Follow:    Available
Rollback:       Unsupported
Created:        2012-05-02 21:54 UTC
Maintenance:    not required (Mondays 23:00 to Tuesdays 03:00 UTC)
Infrastructure: Legacy
```

Performance analytics

Performance analytics is the visibility suite for Heroku Postgres. It enables you to monitor the performance of your database and diagnose potential problems. It consists of several components:

Expensive queries

The leading cause of poor database performance is unoptimized queries. The list of your most expensive queries, available through data.heroku.com (https://data.heroku.com), helps to identify and understand the queries that take the most time in your database. Full documentation is available here (https://devcenter.heroku.com/articles/expensive-queries).

Logging

If your application/framework emits logs on database access, you can retrieve them through Heroku's log-stream (https://devcenter.heroku.com/articles/logging#log-retrieval):

```
$ heroku logs -t
```

To see logs from the database service itself, use `heroku logs` with the `-p postgres` option to see only the logs from Postgres:

```
$ heroku logs -p postgres -t
```

To minimize impact on database performance, logs are delivered on a best-effort basis.

Read more about **Heroku Postgres log statements (https://devcenter.heroku.com/articles/postgres-logs-errors)**.

pg:diagnose

`pg:diagnose` performs a number of useful health and diagnostic checks that help analyze and optimize database performance. It produces a report that you can share with teammates or Heroku Support.

Before taking any action based on a report, make sure to carefully consider the impact to your database and application.

```
$ heroku pg:diagnose --app sushi
Report 1234abc… for sushi::HEROKU_POSTGRESQL_MAROON_URL
available for one month after creation on 2014-07-03 21:29:40.868968+00

GREEN: Connection Count
GREEN: Long Queries
GREEN: Long Transactions
GREEN: Idle in Transaction
GREEN: Indexes
GREEN: Bloat
GREEN: Hit Rate
GREEN: Blocking Queries
GREEN: Sequences
GREEN: Table Transaction ID Wraparound
GREEN: Database Transaction ID Wraparound
GREEN: Schema Count
GREEN: Load
```

Check: Connection Count

Each Postgres connection requires memory, and database plans have a limit on the number of connections they can accept. If you are using too many connections, consider using a connection pooler such as PgBouncer (https://devcenter.heroku.com/articles/concurrency-and-database-connections#limit-connections-with-pgbouncer) or migrating to a larger plan with more RAM.

Checks: Long Running Queries, Long Transactions, Idle in Transaction

Long-running queries and transactions can cause problems with bloat that prevent auto vacuuming (https://devcenter.heroku.com/articles/managing-vacuum-on-heroku-postgres#vacuuming-a-database) and cause followers to lag behind. They also create locks on your data, which can prevent other transactions from running. Consider killing long-running queries with `pg:kill`.

Backends that are `idle in transaction` are backends that are waiting on the client to finish the transaction, either through `COMMIT;` or `ROLLBACK`. Clients that have improperly disconnected may leave backends in this state, and they should be terminated with `pg:kill --force` if left open.

Check: Indexes

The Indexes check includes three classes of indexes.

Never Used Indexes have not been used (since the last manual database statistics refresh). These indexes are typically safe to drop, unless they are in use on a follower.

Low Scans, High Writes indexes are used, but infrequently relative to their write volume. Indexes are updated on every write, so are especially costly on a high write table. Consider the cost of slower writes against the performance improvements that these indexes provide.

Seldom used Large Indexes are not used often and take up significant space both on disk and in cache (RAM). These indexes may still be important to your application, for example, if they are used by periodic jobs or infrequent traffic patterns.

Index usage is only tracked on the database receiving the query. If you use followers for reads, this check will not account for usage made against the follower and is likely inaccurate.

Check: Bloat

Because Postgres uses MVCC (https://devcenter.heroku.com/articles/postgresql-concurrency) old versions of updated or deleted rows are simply made invisible rather than modified in place. Under normal operation an auto vacuum (https://devcenter.heroku.com/articles/managing-vacuum-on-heroku-postgres#vacuuming-a-database) process goes through and asynchronously cleans these up. However sometimes it cannot work fast enough or otherwise cannot prevent some tables from becoming bloated. High bloat can slow down queries, waste space, and even increase load as the database spends more time looking through dead rows.

You can manually vacuum a table with the `VACUUM (VERBOSE, ANALYZE);` command in `psql`. If this occurs frequently you may want to make autovacuum more aggressive (https://devcenter.heroku.com/articles/managing-vacuum-on-heroku-postgres#automatic-vacuuming-with-autovacuum).

It's important to note that the bloat estimaton calculation may not be accurate for tables that use columns that do not have column statistics, such as `json` columns. This is especially pronounced on Postgres 9.4 and below.

Check: Hit Rate

This checks the overall index hit rate, the overall cache hit rate, and the individual index hit rate per table. Databases with lower cache hit rates perform significantly worse as they have to hit disk instead of reading from memory. Consider migrating to a larger plan (https://devcenter.heroku.com/articles/heroku-postgres-plans#determining-required-cache-size) for low cache hit rates, and adding appropriate indexes for low index hit rates.

The overall cache hit rate is calculated as a ratio of table data blocks fetched from the Postgres buffer cache against the sum of cached blocks and un-cached blocks read from disk. On larger plans, the cache hit ratio may be lower but performance remains constant, as the remainder of the data is cached in memory by the OS rather than Postgres.

The overall index hit rate is calculated as a ratio of index blocks fetched from the Postgres buffer cache against the sum of cached indexed blocks and un-cached index blocks read from disk. On larger plans, the index hit ratio may be lower, but performance remains constant, as the remainder of the index data is cached in memory by the OS rather than Postgres.

The individual index hit rate per table is calculated as a ratio of index scans against a table versus the sum of sequential scans and index scans against the table.

Check: Blocking Queries

Some queries can take locks that block other queries from running. Normally these locks are acquired and released very quickly and do not cause any issues. In pathological situations however some queries can take locks that cause significant problems if held too long. You may want to consider killing the query with `pg:kill`.

Check: Sequences

This looks at 32bit `integer` (aka int4) columns that have associated sequences, and reports on those that are getting close to the maximum value for 32bit ints. You should migrate these columns to 64bit `bigint` (aka int8) columns to avoid overflow. An example of such a migration is `alter table products alter column id type bigint;`. Changing the column type can be an expensive operation, and sufficient planning should be made for this on large tables. There is no reason to prefer `integer` columns over `bigint` columns (aside from composite indexes) on Heroku Postgres due to alignemnt considerations on 64bit systems.

This check will be skipped if there are more than 100 `integer` (int4) columns.

Check: Table Transaction ID Wraparound, Database Transaction ID Wraparound

These checks determine how close individual tables are, or a database is, to transaction ID wraparound (https://www.postgresql.org/docs/current/routine-vacuuming.html#VACUUM-FOR-WRAPAROUND). This is a very rare scenario in which due to autovacuum operations being unable to keep up on very frequently updated tables, these tables are in danger of the transaction ID for that table, or database, wrapping around and resulting in data loss. To prevent this, Postgres will prevent new writes cluster wide until this is resolved, impacting availability. These checks return the table and database names if over 50% of the transaction ID space has been used.

Check: Schema Count

This check counts the number of schema are present in the database, returning a yellow warning for over 19 schema, and a red warning over 50 schema. Postgres performance and the ability to take successful logical backups can be affected significantly with very large numbers of schema, and we recommend to maintain no more than 50 schema.

Check: Load

There are many, many reasons that load can be high on a database: bloat, CPU intensive queries, index building, and simply too much activity on the database. Review your access patterns, and consider migrating to a larger plan (https://devcenter.heroku.com/articles/heroku-postgres-plans) which would have a more powerful processor.

Local setup

- Mac setup (https://devcenter.heroku.com/articles/heroku-postgresql#set-up-postgres-on-mac)
- Windows setup (https://devcenter.heroku.com/articles/heroku-postgresql#set-up-postgres-on-windows)
- Linux setup (https://devcenter.heroku.com/articles/heroku-postgresql#set-up-postgres-on-linux)

Heroku recommends running Postgres locally to ensure parity between environments (http://www.12factor.net/dev-prod-parity). There are several pre-packaged installers for installing PostgreSQL in your local environment.

Once Postgres is installed and you can connect, you'll need to export the DATABASE_URL environment variable for your app to connect to it when running locally:

```
-- for Mac and Linux
$ export DATABASE_URL=postgres://$(whoami)
-- for Windows
$ set DATABASE_URL=postgres://$(whoami)
```

This tells Postgres to connect locally to the database matching your user account name (which is set up as part of installation).

Set up Postgres on Mac

Postgres.app requires Mac OS 10.7 or above.

1. Install Postgres.app (http://postgresapp.com/) and follow setup instructions.
2. Install the postgres CLI tools (http://postgresapp.com/documentation/cli-tools.html).
3. Open up a new terminal window to ensure your changes have been saved.
4. Verify that it worked correctly. The OS X version of `psql` should point to the path containing the `Postgres.app` directory.

If you are using version 9.5, the output looks similar to this:

```
$ which psql
/Applications/Postgres.app/Contents/Versions/latest/bin/psql
```

This command should work correctly:

```
$ createdb
$  psql -h localhost
psql (9.5.2)
Type "help" for help.
=# \q
```

Also verify that the app is set to *automatically* start at login.

PostgreSQL ships with several useful binaries, such as `pg_dump` and `pg_restore`, that you will likely want to use. Add the `/bin` directory that ships with Postgres.app to your PATH (preferably in `.profile`, `.bashrc`, `.zshrc`, or the like to make sure this is set for every terminal session):

```
PATH="/Applications/Postgres.app/Contents/Versions/latest/bin:$PATH"
```

Set up Postgres on Windows

Install Postgres on Windows by using the Windows installer (http://www.enterprisedb.com/products-services-training/pgdownload#windows).

Remember to update your PATH environment variable to add the `bin` directory of your Postgres installation. The directory will be similar to this: `C:\Program Files\PostgreSQL\<VERSION>\bin`. If you forget to update your PATH, commands like `heroku pg:psql` won't work.

Set up Postgres on Linux

Install Postgres via your package manager. The actual package manager command you use will depend on your distribution. The following will work on Ubuntu, Debian, and other Debian-derived distributions:

```
$ sudo apt-get install postgresql
```

If you do not have a package manager on your distribution or the Postgres package is not available, install Postgres on Linux using one of the Generic installers (http://www.enterprisedb.com/products-services-training/pgdownload).

The `psql` client will typically be installed in `/usr/bin`:

```
$ which psql
/usr/bin/psql
```

and the command should work correctly:

```
$ psql
psql (9.3.5)
Type "help" for help.
maciek# \q
```

Using the CLI

Heroku Postgres is integrated directly into the Heroku CLI (https://devcenter.heroku.com/articles/heroku-cli) and offers many helpful commands that simplify common database tasks.

pg:info

To see all PostgreSQL databases provisioned by your application and the identifying characteristics of each (such as database size, status, number of tables, and PG version), use the `heroku pg:info` command:

```
$ heroku pg:info
=== HEROKU_POSTGRESQL_RED
Plan         Standard 0
Status       available
Data Size    82.8 GB
Tables       13
PG Version   9.5.3
Created      2012-02-15 09:58 PDT
=== HEROKU_POSTGRESQL_GRAY
Plan         Standard 2
Status       available
Data Size    82.8 GB
…
```

To continuously monitor the status of your database, pass `pg:info` through the unix watch command (http://en.wikipedia.org/wiki/Watch_(Unix)):

```
$ watch heroku pg:info
```

pg:psql

`psql` is the native **PostgreSQL interactive terminal (http://www.postgresql.org/docs/current/static/app-psql.html)** and is used to execute queries and issue commands to the connected database.

To establish a `psql` session with your remote database, use `heroku pg:psql`.

You must have PostgreSQL **installed on your system** to use `heroku pg:psql`.

```
$ heroku pg:psql
Connecting to HEROKU_POSTGRESQL_RED... done
psql (9.5.3, server 9.5.3)
SSL connection (cipher: DHE-RSA-AES256-SHA, bits: 256)
Type "help" for help.

rd2lk8ev3jt5j50=> SELECT * FROM users;
```

If you have more than one database, specify the database to connect to (just the color works as a shorthand) as the first argument to the command (the database located at `DATABASE_URL` is used by default).

```
$ heroku pg:psql gray
Connecting to HEROKU_POSTGRESQL_GRAY... done
...
```

pg:push and pg:pull

For a more in depth guide on working with backups, read **the import and export guide. (https://devcenter.heroku.com/articles/heroku-postgres-import-export)**

pg:pull

`pg:pull` can be used to pull remote data from a Heroku Postgres database to a database on your local machine. The command looks like this:

```
$ heroku pg:pull HEROKU_POSTGRESQL_MAGENTA mylocaldb --app sushi
```

This command creates a new local database named `mylocaldb` and then pulls data from the database at `DATABASE_URL` from the app `sushi`. To prevent accidental data overwrites and loss, the local database *must not already exist*. You will be prompted to drop an already existing local database before proceeding.

If providing a Postgres user or password for your local DB is necessary, use the appropriate environment variables like so:

```
$ PGUSER=postgres PGPASSWORD=password heroku pg:pull HEROKU_POSTGRESQL_MAGENTA mylocaldb --
```

As with all `pg:*` commands, you can use shorthand database identifiers here. For example, to pull data from `HEROKU_POSTGRESQL_RED` on the app `sushi`, you could run `heroku pg:pull sushi::RED mylocaldb`.

pg:push

`pg:push` pushes data from a local database into a remote Heroku Postgres database. The command looks like this:

```
$ heroku pg:push mylocaldb HEROKU_POSTGRESQL_MAGENTA --app sushi
```

This command takes the local database `mylocaldb` and pushes it to the database at `DATABASE_URL` on the app `sushi`. To prevent accidental data overwrites and loss, the remote database *must be empty*. You will be prompted to `pg:reset` a remote database that is not empty.

Usage of the `PGUSER` and `PGPASSWORD` for your local database is also supported for `pg:push`, just like for the `pg:pull` command.

Troubleshooting

These commands rely on the `pg_dump` and `pg_restore` binaries that are included in a Postgres installation. It is somewhat common, however, for the wrong binaries to be loaded in `$PATH`. Errors such as

```
!    createdb: could not connect to database postgres: could not connect to server: No such
!      Is the server running locally and accepting
!      connections on Unix domain socket "/var/pgsql_socket/.s.PGSQL.5432"?
!
!    Unable to create new local database. Ensure your local Postgres is working and try aga
```

and

```
pg_dump: server version: 9.5.3; pg_dump version: 9.5.3
pg_dump: aborting because of server version mismatch
pg_dump: *** aborted because of error
pg_restore: [archiver] input file is too short (read 0, expected 5)
```

are both often a result of this incorrect `$PATH` problem. This problem is especially common with Postgres.app users, as the post-install step of adding `/Applications/Postgres.app/Contents/MacOS/bin` to `$PATH` is easy to forget.

pg:ps, pg:kill, pg:killall

These commands give you view and control over currently running queries.

The `pg:ps` command queries the `pg_stat_activity` view in Postgres to give a concise view into currently running queries.

```
$ heroku pg:ps
 procpid |         source            |   running_for   | waiting |         query
---------+---------------------------+-----------------+---------+-----------------------
   31776 | psql                      | 00:19:08.017088 | f       | <IDLE> in transaction
   31912 | psql                      | 00:18:56.12178  | t       | select * from hello;
   32670 | Heroku Postgres Data Clip | 00:00:25.625609 | f       | BEGIN READ ONLY; select
(3 rows)
```

The procpid column can then be used to cancel or terminate those queries with `pg:kill`. Without any arguments pg_cancel_backend (http://www.postgresql.org/docs/current/static/functions-admin.html#FUNCTIONS-ADMIN-SIGNAL-TABLE) is called on the query which will attempt to cancel the query. In some situations that can fail, in which case the `--force` option can be used to issue pg_terminate_backend (http://www.postgresql.org/docs/current/static/functions-admin.html#FUNCTIONS-ADMIN-SIGNAL-TABLE) which drops the entire connection for that query.

```
$ heroku pg:kill 31912
 pg_cancel_backend
-------------------
 t
(1 row)

$ heroku pg:kill --force 32670
 pg_terminate_backend
----------------------
 t
(1 row)
```

`pg:killall` is similar to `pg:kill` except it will cancel or terminate every query on your database.

pg:promote

In setups where more than one database is provisioned (common use-cases include a leader/follower high-availability setup (https://devcenter.heroku.com/articles/heroku-postgres-follower-databases) or as part of the database upgrade process (https://devcenter.heroku.com/articles/upgrading-heroku-postgres-databases#upgrading-with-pg-copy)) it is often necessary to promote an auxiliary database to the primary role. This is accomplished with the `heroku pg:promote` command.

```
$ heroku pg:promote HEROKU_POSTGRESQL_GRAY_URL
Promoting HEROKU_POSTGRESQL_GRAY_URL to DATABASE_URL... done
```

`pg:promote` works by setting the value of the `DATABASE_URL` config var (which your application uses to connect to the primary database) to the newly promoted database's URL and restarting your app. The old primary database location is still accessible via its `HEROKU_POSTGRESQL_COLOR_URL` setting.

After a promotion, the demoted database is still provisioned and incurring charges. If it's no longer need you can remove it with `heroku addons:destroy HEROKU_POSTGRESQL_COLOR`.

pg:credentials

Heroku Postgres provides convenient access to the credentials and location of your database should you want to use a GUI to access your instance.

The database name argument must be provided with `pg:credentials:url` command. Use `DATABASE` for your primary database.

```
$ heroku pg:credentials:url DATABASE
Connection info string:
   "dbname=dee932clc3mg8h host=ec2-123-73-145-214.compute-1.amazonaws.com port=6212 user=use
```

It is a good security practice to rotate the credentials for important services on a regular basis. On Heroku Postgres this can be done with `heroku pg:credentials:rotate`.

```
$ heroku pg:credentials:rotate HEROKU_POSTGRESQL_GRAY_URL
```

When you issue this command, new credentials are created for your database and the related config vars on your Heroku application are updated. However, on Standard, Premium, and Enterprise tier databases (https://devcenter.heroku.com/articles/heroku-postgres-plans#standard-tier) the old credentials are not removed immediately. All of the open connections remain open until the currently running tasks complete, then those credentials are updated. This is to make sure that any background jobs or other workers running on your production environment aren't abruptly terminated, which could potentially leave the system in an inconsistent state.

pg:reset

The PostgreSQL user your database is assigned doesn't have permission to create or drop databases. To drop and recreate your database use `pg:reset`.

```
$ heroku pg:reset DATABASE
```

Heroku Postgres & SSL

Most clients will connect over SSL by default, but on occasion it is necessary to set the `sslmode=require` parameter on a Postgres connection. Please add this parameter in code rather than editing the config var directly. Please check you are enforcing use of SSL especially if you are using Java or Node.js clients.

Connecting in Java

There are a variety of ways to create a connection to a Heroku Postgres database, depending on the Java framework in use. In most cases, the environment variable `JDBC_DATABASE_URL` can be used directly as described in the article Connecting to Relational Databases on Heroku with Java (https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#using-the-jdbc_database_url). Here is an example:

```
private static Connection getConnection() throws URISyntaxException, SQLException {
    String dbUrl = System.getenv("JDBC_DATABASE_URL");
    return DriverManager.getConnection(dbUrl);
}
```

When it is not possible to use the JDBC URL (usually because custom buildpack is being used), you must use the `DATABASE_URL` environment URL to determine connection information. Some examples are provided below.

By default, Heroku will attempt to enable SSL for the PostgreSQL JDBC driver by setting the property `sslmode=require` globally. However, if you are building the JDBC URL yourself (such as by parsing the `DATABASE_URL`) then we recommend explicitly adding this parameter.

It is also important that you use a version of the Postgres JDBC driver verion 9.2 or greater. For example, in Maven add this to your `pom.xml`:

```
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>42.2.1</version>
</dependency>
```

Examples of all outlined connection methods here are available on GitHub at: **https://github.com/heroku/devcenter-java-database (https://github.com/heroku/devcenter-java-database)**

JDBC

Create a JDBC connection to Heroku Postgres by parsing the `DATABASE_URL` environment variable.

```
private static Connection getConnection() throws URISyntaxException, SQLException {
    URI dbUri = new URI(System.getenv("DATABASE_URL"));

    String username = dbUri.getUserInfo().split(":")[0];
    String password = dbUri.getUserInfo().split(":")[1];
    String dbUrl = "jdbc:postgresql://" + dbUri.getHost() + ':' + dbUri.getPort() + dbUri.ge

    return DriverManager.getConnection(dbUrl, username, password);
}
```

Spring/XML

This snippet of Spring XML configuration will setup a `BasicDataSource` from the `DATABASE_URL` and can then be used with Hibernate, JPA, etc:

```
<bean class="java.net.URI" id="dbUrl">
    <constructor-arg value="#{systemEnvironment['DATABASE_URL']}"/>
</bean>

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="url" value="#{ 'jdbc:postgresql://' + @dbUrl.getHost() + ':' + @dbUrl.ge
    <property name="username" value="#{ @dbUrl.getUserInfo().split(':')[0] }"/>
    <property name="password" value="#{ @dbUrl.getUserInfo().split(':')[1] }"/>
</bean>
```

Spring/Java

Alternatively you can use Java for configuration of the `BasicDataSource` in Spring:

```
@Configuration
public class MainConfig {

    @Bean
    public BasicDataSource dataSource() throws URISyntaxException {
        URI dbUri = new URI(System.getenv("DATABASE_URL"));

        String username = dbUri.getUserInfo().split(":")[0];
        String password = dbUri.getUserInfo().split(":")[1];
        String dbUrl = "jdbc:postgresql://" + dbUri.getHost() + ':' + dbUri.getPort() + dbU

        BasicDataSource basicDataSource = new BasicDataSource();
        basicDataSource.setUrl(dbUrl);
        basicDataSource.setUsername(username);
        basicDataSource.setPassword(password);

        return basicDataSource;
    }
}
```

The `DATABASE_URL` for the Heroku Postgres add-on follows this naming convention:

```
postgres://<username>:<password>@<host>/<dbname>
```

However the Postgres JDBC driver uses the following convention:

```
jdbc:postgresql://<host>:<port>/<dbname>?sslmode=require&user=<username>&password=<password>
```

Notice the additional `ql` at the end of `jdbc:postgresql`? Due to this difference you will need to hardcode the scheme to `jdbc:postgresql` in your Java class or your Spring XML configuration.

Remote connections

You can connect to your Heroku Postgres database remotely for maintenance and debugging purposes. However, doing so requires that you use an SSL connection. Your JDBC connection URL will need to include the following URL parameter:

```
sslmode=require
```

If you leave off `sslmode=require` you will get a connection error when attempting to connect to production-tier databases.

Note: it is important to add this parameter in code rather than editing the config var directly. Various automated events such as failover can change the config var, and edits there would be lost.

Click here for more information see the Dev Center article on Connecting to Relational Databases on Heroku with Java (https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#connecting-to-a-database-remotely).

Connecting in Ruby

To use PostgreSQL as your database in Ruby applications you will need to include the `pg` gem in your `Gemfile`.

```
gem 'pg'
```

Run `bundle install` to download and resolve all dependencies.

If you are using the `pg` gem to connect to your Postgres database from a Heroku dyno, and have not specified an `sslmode` in your configuration or code, the gem will default to `sslmode: prefer`: this means that your connections will work if SSL use is enforced on your Postgres database.

Connecting in Rails

When Rails applications are deployed to Heroku a `database.yml` file is automatically generated (https://devcenter.heroku.com/articles/ruby-support#build-behavior) for your application. That configures ActiveRecord to use a PostgreSQL connection and to connect to the database located at `DATABASE_URL`. This behavior is only needed up to Rails 4.1. Any later version contains direct support for specifying a connection URL and configuration in the `database.yml` so we do not have to overwrite it.

To use PostgreSQL locally with a Rails app your `database.yml` should contain the following configuration:

```
development:
  adapter: postgresql
  host: localhost
  username: user
  database: app-dev
```

Connecting in JRuby

To use PostgreSQL as your database in JRuby applications you will need to include the `activerecord-jdbcpostgresql-adapter` gem in your `Gemfile`.

```
gem 'activerecord-jdbcpostgresql-adapter'
```

Run `bundle install` to download and resolve all dependencies.

If using Rails, follow the instructions for Connecting with Rails (https://devcenter.heroku.com/articles/heroku-postgresql#connecting-in-rails).

Connecting in Python

To use PostgreSQL as your database in Python applications you will need to use the `psycopg2` package.

```
$ pip install psycopg2-binary
```

And use this package to connect to `DATABASE_URL` in your code.

```
import os
import psycopg2

DATABASE_URL = os.environ['DATABASE_URL']

conn = psycopg2.connect(DATABASE_URL, sslmode='require')
```

Connecting with Django

Install the `dj-database-url` package using `pip`.

```
$ pip install dj-database-url
```

Be sure to add `psycopg2-binary` and `dj-database-url` to your `requirements.txt` file as well.

Then add the following to the bottom of `settings.py`:

```
import dj_database_url
DATABASES['default'] = dj_database_url.config(conn_max_age=600, ssl_require=True)
```

This will parse the values of the `DATABASE_URL` environment variable and convert them to something Django can understand.

Connecting in Go

Go apps can connect to Heroku-Postgres by providing the pq (https://godoc.org/github.com/lib/pq#hdr-Connection_String_Parameters) Postgres database driver to their query interface of choice (such as the standard database/sql (https://godoc.org/database/sql)). Your app will use the query interface, rather than using the driver directly.

Standard usage (database/sql)

```
$ cd <app>
$ dep ensure -add github.com/lib/pq
```

```
import (
  "database/sql"
  _ "github.com/lib/pq"
)
...
func main() {
  db, err := sql.Open("postgres", os.Getenv("DATABASE_URL"))
  if err != nil {
    log.Fatal(err)
  }
  ...
}
```

SSL is required to connect to Heroku-Postgres. pq automatically sets **sslmode=require (https://godoc.org/github.com/lib/pq#hdr-Connection_String_Parameters)**, but if you use another library, you may need to configure ssl explicitly.

Beyond the standard lib

For lower-level access to Postgres, you can use pgx (https://godoc.org/github.com/jackc/pgx).

For time-saving extensions to database/sql, you can use sqlx (http://jmoiron.github.io/sqlx/).

Connecting in PHP

General considerations

If a framework or library cannot natively handle database URLs, and instead requires separate arguments for user, pass, host, port, database name and so on, then the `parse_url()` (http://php.net/parse_url) function can be used to parse the `DATABASE_URL` environment variable and establish a connection using the extracted details.

> The leading slash needs to be trimmed from the path component, as that contains the database name.

```
$db = parse_url(getenv("DATABASE_URL"));
$db["path"] = ltrim($db["path"], "/");
```

The resulting associative array will contain the information from the URL, as documented (http://php.net/parse_url), with the database name available through the "`path`" key.

Connecting with the pgsql extension

The pgsql (http://php.net/pgsql) extension passes connection strings directly to the underlying `libpq` library, which supports URL-style connection strings, so the `DATABASE_URL` environment variable can be used directly:

```
$conn = pg_connect(getenv("DATABASE_URL"));
```

Connecting with PDO

A DSN (http://php.net/manual/en/ref.pdo-pgsql.connection.php) needs to be constructed to connect using PDO (http://php.net/manual/en/book.pdo.php):

```
$db = parse_url(getenv("DATABASE_URL"));

$pdo = new PDO("pgsql:" . sprintf(
    "host=%s;port=%s;user=%s;password=%s;dbname=%s",
    $db["host"],
    $db["port"],
    $db["user"],
    $db["pass"],
    ltrim($db["path"], "/")
));
```

Connecting with Laravel

The `config/database.php` file returns an array of database connection info to the framework; it can simply be amended to call `parse_url()` on the `DATABASE_URL` environment variable first, and return the extracted data:

```
$DATABASE_URL = parse_url(getenv("DATABASE_URL"));

return [

    // …

    'connections' => [

        // …

        'pgsql' => [
            'driver' => 'pgsql',
            'host' => $DATABASE_URL["host"],
            'port' => $DATABASE_URL["port"],
            'database' => ltrim($DATABASE_URL["path"], "/"),
            'username' => $DATABASE_URL["user"],
            'password' => $DATABASE_URL["pass"],
            'charset' => 'utf8',
            'prefix' => '',
            'schema' => 'public',
            'sslmode' => 'require',
        ],

        // …

    ],

    // …

];
```

Connecting with Symfony 3

The `DATABASE_URL` environment variable can be referenced (https://devcenter.heroku.com/articles/deploying-symfony3#environment-variables) in `config.yml`; Symfony's DoctrineBundle (https://symfony.com/doc/3.4/reference/configuration/doctrine.html#reference-dbal-configuration) will then automatically parse the URL's contents.

Connecting with Symfony 4

Symfony 4 will automatically pick up (https://symfony.com/doc/current/doctrine/dbal.html) the `DATABASE_URL` environment variable without further configuration.

Connecting in Node.js

Install the `pg` NPM module as a dependency:

```
$ npm install pg
```

Then, connect to `process.env.DATABASE_URL` when your app initializes:

```
const { Client } = require('pg');

const client = new Client({
  connectionString: process.env.DATABASE_URL,
  ssl: true,
});

client.connect();

client.query('SELECT table_schema,table_name FROM information_schema.tables;', (err, res) =>
  if (err) throw err;
  for (let row of res.rows) {
    console.log(JSON.stringify(row));
  }
  client.end();
});
```

Note: SSL connections are required to connect Heroku Postgres

Connection permissions

Heroku Postgres users are granted all non-superuser permissions on their database. These include `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, `REFERENCES`, `TRIGGER`, `CREATE`, `CONNECT`, `TEMPORARY`, `EXECUTE`, and `USAGE` and limited `ALTER` and `GRANT` permissions.

Heroku runs the SQL below to create a user and database for you.

```
CREATE ROLE user_name;
ALTER ROLE user_name WITH LOGIN PASSWORD 'password' NOSUPERUSER NOCREATEDB NOCREATEROLE;
CREATE DATABASE database_name OWNER user_name;
REVOKE ALL ON DATABASE database_name FROM PUBLIC;
GRANT CONNECT ON DATABASE database_name TO user_name;
GRANT ALL ON DATABASE database_name TO user_name;
```

Multiple schemas

Heroku Postgres supports multiple schemas and does not place any limits on the number of schemas you can create.

The most common use case for using multiple schemas in a database is building a software-as-a-service application wherein each customer has their own schema. While this technique seems compelling, we strongly recommend against it as it has caused numerous cases of operational problems. For instance, even a moderate number of schemas (> 50) can severely impact the performance of Heroku's database snapshots tool, **PG Backups (https://devcenter.heroku.com/articles/heroku-postgres-backups)**.

External connections (ingress)

In addition to being available to the Heroku runtime, Heroku Postgres databases can be accessed directly by clients running on your local computer or elsewhere.

All connections require SSL: `sslmode=require`.

You can retrieve the PG connection string in one of two ways. `heroku pg:credentials` is discussed above:

```
$ heroku pg:credentials DATABASE
Connection info string:
   "dbname=dee932clc3mg8h host=ec2-123-73-145-214.compute-1.amazonaws.com port=6212 user=use
```

Also, the connection string is exposed as a config var for your app:

```
$ heroku config | grep HEROKU_POSTGRESQL
HEROKU_POSTGRESQL_YELLOW_URL: postgres://user3123:passkja83kd8@ec2-117-21-174-214.compute-1
```

Migrating between plans

See this detailed guide on updating and migrating between database plans (https://devcenter.heroku.com/articles/updating-heroku-postgres-databases).

Data Residency

When a database gets provisioned, the data associated with that database is stored within the region in which it's created. However, a number of services that are ancillary to Heroku Postgres as well as the systems that manage the fleet of databases may not be located within the same region as the provisioned databases. Here are some

- Postgres Continuous Protection (https://devcenter.heroku.com/articles/heroku-postgres-data-safety-and-continuous-protection) for disaster recovery stores the base backup and write-ahead logs in the same region that the database is located.
- Application logs are routed to Logplex (https://devcenter.heroku.com/articles/logplex), which is hosted in the US. In addition to logs from your application, this includes System logs (https://devcenter.heroku.com/articles/logging#types-of-logs) and Heroku Postgres logs from any database attached to your application.
- Logging of Heroku Postgres queries and errors can be blocked by using the `--block-logs` flag when creating the database with `heroku addons:create heroku-postgres:...`.
- PG Backup (https://devcenter.heroku.com/articles/heroku-postgres-backups) snapshots are stored in the US.
- Dataclips are stored in the US.

Blocking Logs

At add-on creation time, a flag can be passed to prevent logging of queries that get run against the database. If this option is turned on, it cannot be turned off after the database has been provisioned. If you need to turn it off after it has been turned on, a migration to a new database will be required.

> By blocking the queries in the logs, this will reduce Heroku's ability to assist in debugging applications and application performance.

```
$ heroku addons:create heroku-postgresql:standard-0 -a sushi --block-logs
```

Removing the add-on

In order to destroy your Heroku Postgres database you will need to remove the add-on.

```
$ heroku addons:destroy heroku-postgresql:hobby-dev
```

If you have two databases of the same type you will need to remove the add-on using its config var name. For example, to remove the `HEROKU_POSTGRESQL_GRAY_URL`, you would run:

```
heroku addons:destroy HEROKU_POSTGRESQL_GRAY
```

If the removed database was the same one used in `DATABASE_URL`, that `DATABASE_URL` config var will also be unset on the app.

Databases cannot be reconstituted after being destroyed. Please take a snapshot of the data beforehand using **PG Backups (https://devcenter.heroku.com/articles/heroku-postgres-backups)** or by **exporting the data (https://devcenter.heroku.com/articles/heroku-postgres-import-export#export)**

Support

All Heroku Postgres support and runtime issues should be submitted via one of the Heroku Support channels (https://devcenter.heroku.com/articles/support-channels).