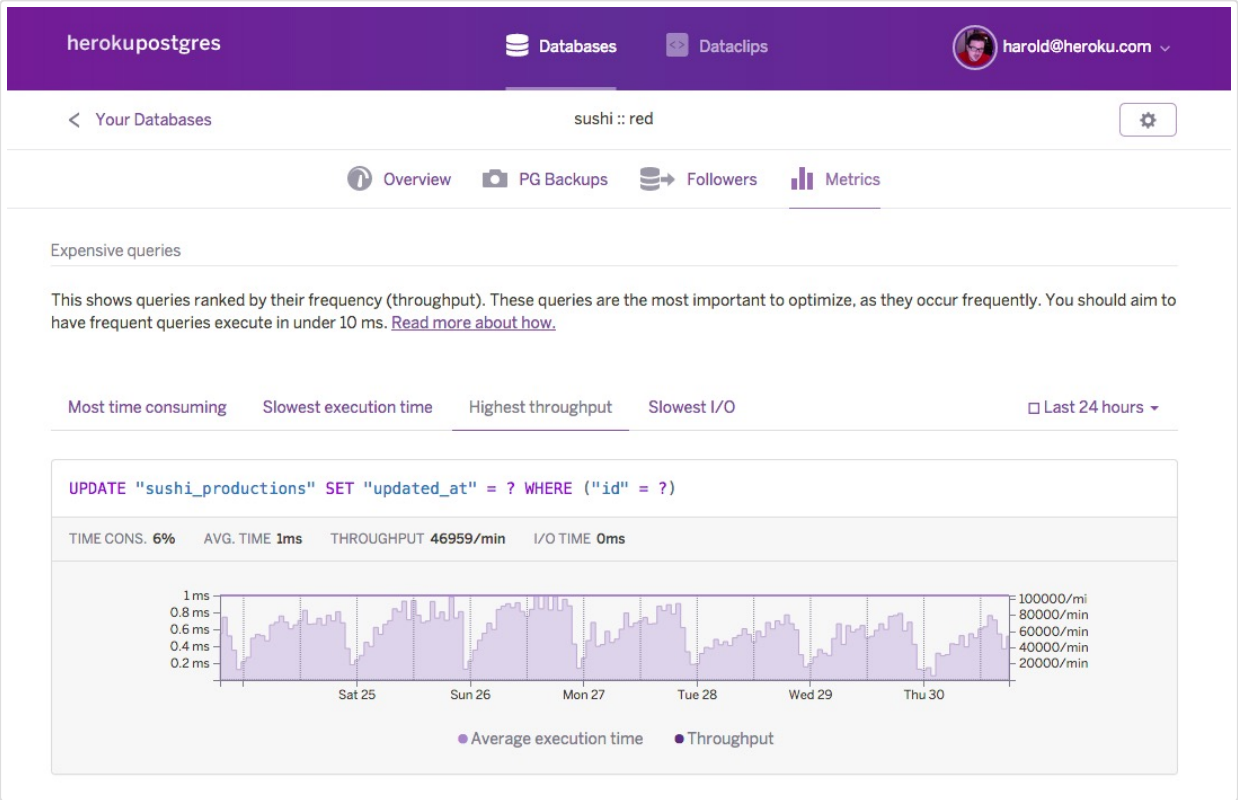
Heroku Postgres

Last updated 14 October 2016

Table of Contents

- Provisioning the add-on
- Version support and legacy infrastructure
- Performance analytics
- Local setup
- Using the CLI
- Heroku Postgres & SSL
- Connecting in Java
- Connecting in Ruby
- Connecting in JRuby
- Connecting in Python
- Connecting in Go
- Connecting in Node.js
- Connection permissions
- External connections (ingress)
- Migrating between plans
- Removing the add-on
- Support

Heroku Postgres (https://elements.heroku.com/addons/heroku-postgresql) is the SQL database service run by Heroku that is provisioned and managed as an add-on (https://elements.heroku.com/addons). Heroku Postgres is accessible from any language with a PostgreSQL driver including all languages and frameworks supported by Heroku: Java, Ruby, Python, Scala, Play, Node.js, PHP, Go, and Clojure.



In addition to a variety of management commands available via the Heroku CLI, Heroku Postgres features a web dashboard (https://postgres.heroku.com/databases), the ability to create dataclips (https://postgres.heroku.com/blog/past/2012/1/31/simple_data_sharing_with_data_clips/) and several additional services on top of a fully managed database service.

Provisioning the add-on

Many buildpacks (https://devcenter.heroku.com/articles/buildpacks) (what compiles your application into a runnable entity on Heroku) automatically provision a Heroku Postgres instance for you. Your language's buildpack documentation (https://devcenter.heroku.com/categories/language-support) will specify if any add-ons are automatically provisioned. Additionally, you can use `heroku addons` to see if your application already has a database provisioned and what plan it is.

```
$ heroku addons | grep -i POSTGRES
heroku-postgresql:hobby-dev  HEROKU_POSTGRESQL_RED
```

In order for Heroku to manage this add-on for you and respond to a variety of operational situations, the value of this config var may change at any time. Relying on it outside your Heroku app may prove problematic as you will have to re-copy the value when it changes.

If your application doesn't yet have a database provisioned, or you wish to upgrade your existing database (https://devcenter.heroku.com/articles/upgrading-heroku-postgres-databases#upgrade-with-pg-copy-default) or create a master/slave setup (https://devcenter.heroku.com/articles/heroku-postgres-follower-databases), you can create a new database using the CLI.

Create a new database

Heroku Postgres offers a variety of plans, spread across different tiers of service (https://devcenter.heroku.com/articles/heroku-postgres-plans#plan-tiers): hobby, standard, premium, and enterprise. For a list of available plans, see Choosing the Right Heroku Postgres Plan (https://devcenter.heroku.com/articles/heroku-postgres-plans). If you eventually outgrow your initial plan, you can upgrade databases (https://devcenter.heroku.com/articles/upgrading-heroku-postgres-databases#upgrade-with-pg-copy-default).

Heroku Postgres can be attached to a Heroku application via the CLI using this command: `heroku addons:create heroku-postgresql:<PLANNAME>` .

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
Adding heroku-postgresql:hobby-dev to sushi... done, v69 (free)
Attached as HEROKU_POSTGRESQL_RED
Database has been created and is available
```

To provision a hobby-basic (https://devcenter.heroku.com/articles/heroku-postgres-plans#hobby-tier) plan database:

```
$ heroku addons:create heroku-postgresql:hobby-basic
```

Depending on the plan, some databases can take up to 5 minutes to become available. Use pg:wait to track their status:

```
$ heroku pg:wait
Waiting for database HEROKU_POSTGRESQL_RED... done
```

Once Heroku Postgres has been added a `DATABASE_URL` or `HEROKU_POSTGRESQL_COLOR_URL` setting will be available in the app configuration and will contain the URL used to access the newly provisioned Heroku Postgres service. `DATABASE_URL` will be given to the database if it is the first one for the application, otherwise the `HEROKU_POSTGRESQL_COLOR_URL` will be created. This can be confirmed using the `heroku config` command.

```
$ heroku config -s | grep HEROKU_POSTGRESQL
HEROKU_POSTGRESQL_RED_URL=postgres://user3123:passkja83kd8@ec2-117-21-174-214.compute-1.amazonaws.com:6
```

You can choose the alias that the add-on uses on the application using the `--as` flag. This will affect the name of the variable the add-on adds to the application:

```
$ heroku addons:create heroku-postgresql:hobby-dev --as USERS_DB
Adding heroku-postgresql:hobby-dev to sushi... done, v69 (free)
Attached as USERS_DB
Database has been created and is available
$ heroku config -s | grep USERS_DB
USERS_DB_URL=postgres://user3123:passkja83kd8@ec2-117-21-174-214.compute-1.amazonaws.com
```

Establish primary DB

Heroku recommends using the `DATABASE_URL` config var to store the location of your primary database. In single-database setups your new database will have already been assigned to `DATABASE_URL` .

On apps with multiple databases, you can set the primary database like this:

```
$ heroku pg:promote HEROKU_POSTGRESQL_RED
Promoting HEROKU_POSTGRESQL_RED_URL to DATABASE_URL... done
```

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.

Sharing Heroku Postgres between applications

You can share one Heroku Postgres between multiple applications.

```
$ 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, v10
```

The database will be attached with a color (in this example `HEROKU_POSTGRESQL_BRONZE` but it will change each time).

The shared database will not necessarily be the default database on any apps that it is shared with. To promote the shared database to be the primary database, use the `pg:promote` command with the color, or the addon name, on each of the apps where you want it to be the default database:

```
$ heroku pg:promote HEROKU_POSTGRESQL_BRONZE --app sushi
Ensuring an alternate alias for existing DATABASE... done, HEROKU_POSTGRESQL_SILVER
Promoting postgresql-addon-name to DATABASE_URL on sushi... done
```

```
$ heroku pg:promote postgresql-addon-name --app sushi
Ensuring an alternate alias for existing DATABASE... done, HEROKU_POSTGRESQL_SILVER
Promoting postgresql-addon-name to DATABASE_URL on sushi... done
```

Version support and legacy infrastructure

PostgreSQL 9.1 is no longer supported. No new provisions are allowed.

The version of Postgres you want to run can be specified by using the `--version` flag within the Heroku CLI.

The PostgreSQL project releases new major versions on a yearly basis. Each major version, once released, will be supported shortly after by Heroku Postgres. Heroku Postgres will support at least 3 major versions at a given time. Currently supported versions are:

- 9.6
- 9.5 (default)
- 9.4
- 9.3
- 9.2

PostgreSQL 9.6 support is in beta right now. You cannot provision hobby tier 9.6 databases, only **standard-0 and above (https://devcenter.heroku.com/articles/heroku-postgres-plans#plan-tiers)**. For production use we recommend PostgreSQL 9.5

By supporting at least 3 major versions, users will be required to upgrade once every three years. However, you can upgrade at any point earlier to gain the benefits of the latest version.

Heroku will also occasionally deprecate old versions of our infrastructure (Legacy Infrastructure). We typically do this if the operating system running beneath the database will no longer receive security update, if support for the OS is no longer practical due to age (if required packages and patches are no longer available or difficult to support), or if the server instances are significantly different from our 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
```

Migration of deprecated databases

When support ends for a given Postgres version or legacy infrastructure, Heroku will provide at least 3 months notification. Databases and infrastructure will be automatically migrated to the latest version when support ends. This automatic migration requires database and application downtime. Heroku **highly recommends** that customers perform the upgrade themselves (https://devcenter.heroku.com/articles/upgrading-heroku-postgres-databases) prior to support ending so that they may test compatibility, have time to plan for unforeseen issues, and are able to migrate the database on their own schedule.

Performance analytics

Performance analytics is the visibility suite for Heroku Postgres. It enables you to monitor the performance of your database and to 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 postgres.heroku.com (https://postgres.heroku.com/databases), 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 will be able to 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 you can also use `heroku logs` but with the `-p postgres` flag indicating that you only want to see the logs from Postgres.

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

In order to have minimal 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)** here.

pg:diagnose

`pg:diagnose` performs a number of useful health and diagnostic checks that help analyze and optimize the performance of a database. The report that can be shared with others on your team or with Heroku Support.

Before taking any action based on a report, be 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: Idle in Transaction
GREEN: Indexes
GREEN: Bloat
GREEN: Hit Rate
GREEN: Blocking Queries
GREEN: Load
GREEN: Sequences
```

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 you may want to 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, 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 causes followers to lag behind. They also create locks on your data which can prevent other transactions from running. You may want to consider killing the long running query with `pg:kill`.

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).

Check: Hit Rate

This checks the overall index hit rate, the overall cache hit rate, and the individual index hit rate per table. It is very important to keep hit rates in the 99+% range. Databases with lower 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.

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: 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.

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;`. 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.

Local setup

Heroku recommends running the same database locally during development as in production (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. E.g.:

```
$ export 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

Install Postgres.app (http://postgresapp.com/) and follow documentation (http://postgresapp.com/documentation). Note that Postgres.app requires Mac OS 10.7 or above. Once installed verify that it worked correctly. The OS X version of `psql` should point to the path containing the `Postgres.app` directory. For example if you are using version 9.5, the output will look similar to this:

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

and this command should work correctly:

```
$  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. Go ahead and add the `/bin` directory that ships with Postgres.app to your PATH (preferably in `.profile`, `.bashrc`, `.zshrc`, or the like to make sure this gets 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 and offers several commands that automate many common tasks associated with managing a database-backed application.

pg:info

To see all PostgreSQL databases provisioned by your application and the identifying characteristics of each (db size, status, number of tables, PG version, creation date etc…) 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

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 will create a new local database named “mylocaldb” and then pull data from database at `DATABASE_URL` from the app “sushi”. In order to prevent accidental data overwrites and loss, the local database *must not 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 this:

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

Note: like all `pg:*` commands you can use the shorthand identifiers here, so to pull data from `HEROKU_POSTGRESQL_RED` on the app “sushi” you could do `heroku pg:pull sushi::RED mylocaldb`.

pg:push

Like pull but in reverse, `pg:push` will push 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 will take the local database “mylocaldb” and push it to the database at `DATABASE_URL` on the app “sushi”. In order to prevent accidental data overwrites and loss, the remote database *must be empty*. You will be prompted to `pg:reset` an already 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 commands.

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 file or dir
!      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 again.
```

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_statements` table 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 'hi'
(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 master/slave 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#upgrade-with-pg-copy-default)) 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` command. Use `DATABASE` for your primary database.

```
$ heroku pg:credentials DATABASE
Connection info string:
   "dbname=dee932clc3mg8h host=ec2-123-73-145-214.compute-1.amazonaws.com port=6212 user=user3121 passw
```

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 --reset`.

```
$ heroku pg:credentials HEROKU_POSTGRESQL_GRAY_URL --reset
```

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

Heroku Postgres databases created on the Common Runtime requires the use of 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. Note: it is important to add this parameter in code rather than editing the config var directly.

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. It is not recommended to disable this, but you may do so by parsing the `DATABASE_URL` manually and add the URL parameter `sslmode=disable` to the resulting JDBC URL.

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>9.4.1208</version>
</dependency>
```

If you use a version earlier than 9.2, which is not recommended, you must add the following parameters to your JDBC URL if you are using a Standard or Premium database:

```
ssl=true&sslfactory=org.postgresql.ssl.NonValidatingFactory
```

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.getPath();

    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.getPo
    <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() + dbUri.getPath

        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>?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.

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.

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
$ pip freeze > requirements.txt
```

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

```
import os
import psycopg2
import urlparse

urlparse.uses_netloc.append("postgres")
url = urlparse.urlparse(os.environ["DATABASE_URL"])

conn = psycopg2.connect(
    database=url.path[1:],
    user=url.username,
    password=url.password,
    host=url.hostname,
    port=url.port
)
```

Connecting with Django

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

```
$ pip install dj-database-url
$ pip freeze > requirements.txt
```

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

```
import dj_database_url
DATABASES['default'] =  dj_database_url.config()
```

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

Connecting in Go

To use Postgres as your database in Go applications you need to use one of the Postgres Go drivers by importing them into your code and using godep to vendor your chosen driver into your code base.

Using github.com/lib/pq

```
$ cd <app>
$ go get -u github.com/lib/pq
```

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

```
$ godep save -r ./...
$ git add -A
```

Additional documentation can be found in the Godocs (http://godoc.org/github.com/lib/pq).

Using github.com/jackc/pgx

```
$ cd <app>
$ go get -u github.com/jackc/pgx
```

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

```
$ godep save -r ./...
$ git add -A
```

Additional documentation can be found on the pgx Godoc page (https://godoc.org/github.com/jackc/pgx) and the pgx stdlib Godoc page (https://godoc.org/github.com/jackc/pgx/stdlib).

Connecting in Node.js

Install the `pg` NPM module as a dependency:

```
$ npm install --save --save-exact pg
```

In your app, configure the module to default to SSL connections. Then, connect to `DATABASE_URL` when your app initializes:

```
var pg = require('pg');

pg.defaults.ssl = true;
pg.connect(process.env.DATABASE_URL, function(err, client) {
  if (err) throw err;
  console.log('Connected to postgres! Getting schemas...');

  client
    .query('SELECT table_schema,table_name FROM information_schema.tables;')
    .on('row', function(row) {
      console.log(JSON.stringify(row));
    });
});
```

Note: SSL connections are required for Heroku Postgres on the Common Runtime. Also, the call to `pg.defaults.ssl` must be done before any connection to the database is created.

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`.

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

You cannot create or modify databases and roles on Heroku Postgres. The SQL below is for reference only.

```
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 database_user;
GRANT ALL ON DATABASE database_name TO database_user;
```

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=user3121 passw
```

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

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

Migrating between plans

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

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).