# Codified PostgreSQL Schema

## Idempotent Schema Migrations for Developers and DBAs

**Sean Chittenden**

*Engineering*

seanc@joyent.com

@SeanChittenden

## Getting Started

- Terraform - terraform.io
  - A basic understanding of a Directed Acyclic Graph
- PostgreSQL - postgresql.org
  - A database and schema to work with
- An appreciation for version controlled software
  - Pick your poison: git, hg, fossil

# Talk Agenda

- Bootstrapping a database by hand
- Bootstrapping a Terraform repository
- Importing a database into Terraform
- Iterating the design of a schema in Terraform
- Explanation of what Terraform is doing under the hood
- Tips to avoid having Terraform become Terrorform

# Bootstrapping a Database

```
$ psql
postgres=# CREATE ROLE pgcon WITH PASSWORD't:<fTuz(vqg?YLY+pASY#}KO3(*&@6' LOGIN;
CREATE ROLE
postgres=# CREATE DATABASE pgcon OWNER pgcon;
CREATE DATABASE
```

# Codified Database Management Explained

```
$ psql -c "CREATE ROLE pgcon WITH PASSWORD't:<fTuz(vqg?YLY+pASY#}KO3(*&@6' LOGIN"
CREATE ROLE
$ psql -c "CREATE DATABASE pgcon OWNER pgcon"
CREATE DATABASE
```

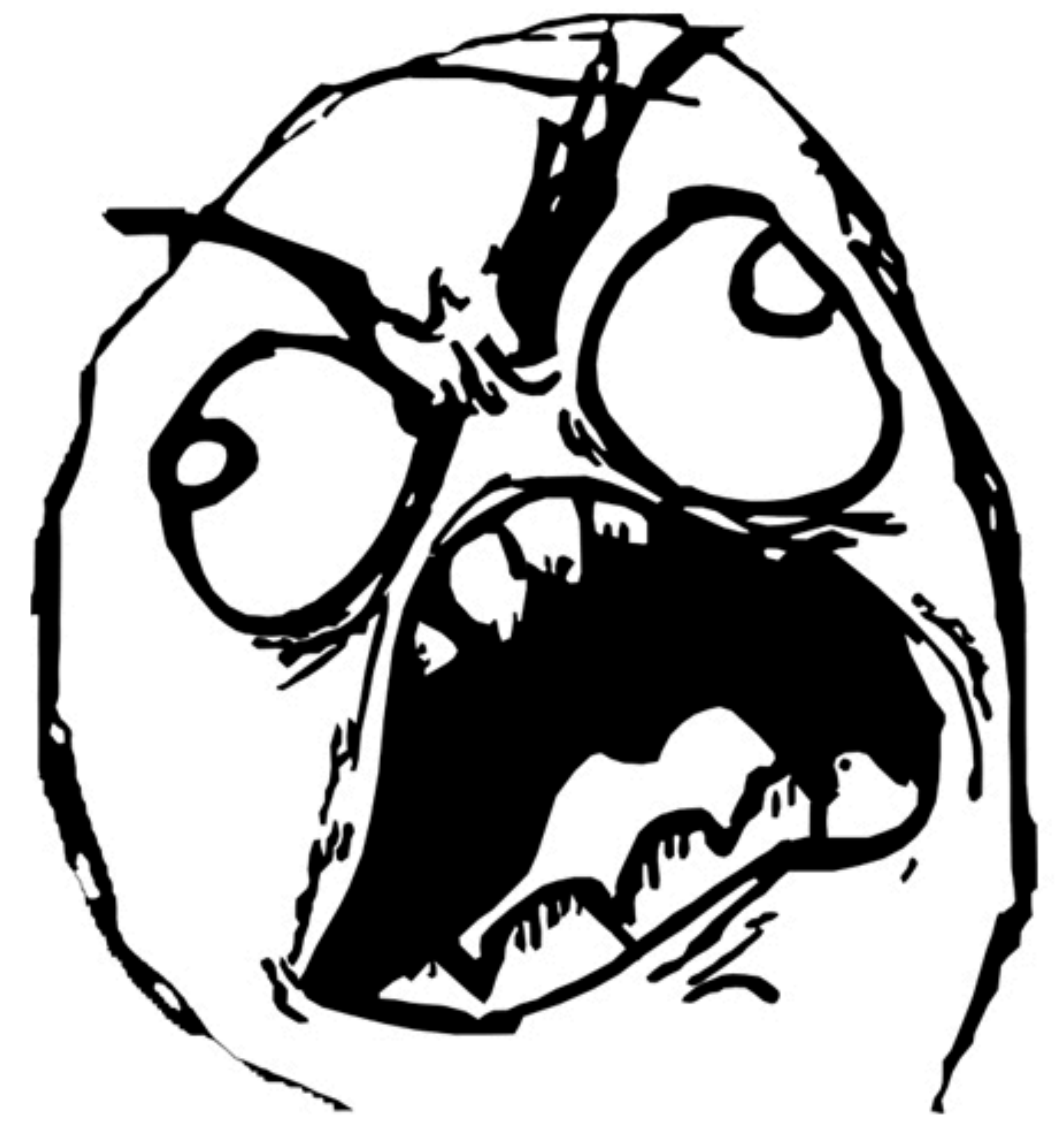
# Codified Database Management Explained

```
$ cat my-db.sh
#!/bin/sh --
set -e
psql -c "CREATE ROLE pgcon WITH PASSWORD't:<fTuz(vqg?YLY+pASY#}KO3(*&@6' LOGIN"
psql -c "CREATE DATABASE pgcon OWNER pgcon"
$ git add my-db.sh
$ git commit -m "psssh!  Automation is e-z"
$ sh my-db.sh
CREATE ROLE
CREATE DATABASE
$ sh my-db.sh
ERROR:  role "pgcon" already exists
```

# Idempotent Database Schemas

Code (and schema) versioning is fundamentally easy because it is internally stateless.

**HACKER NEWS,**
**Strategies for Database Success**



# Codified Database Management, Fixed It

```
$ cat my-db.sh
#!/bin/sh --
set -e
psql -c "DROP DATABASE IF EXISTS pgcon"
psql -c "DROP ROLE IF EXISTS pgcon"
psql -c "CREATE ROLE pgcon WITH PASSWORD't:<fTuz(vqg?YLY+pASY#}KO3(*&@6' LOGIN"
psql -c "CREATE DATABASE pgcon OWNER pgcon"
$ git add my-db.sh
$ git commit -m "fixed it #yolo"
$ sh my-db.sh
NOTICE:  database "pgcon1" does not exist, skipping
DROP DATABASE
NOTICE:  role "pgcon1" does not exist, skipping
DROP ROLE
CREATE ROLE
CREATE DATABASE
$ sh my-db.sh
NOTICE:  database "pgcon1" does not exist, skipping
DROP DATABASE
NOTICE:  role "pgcon1" does not exist, skipping
DROP ROLE
CREATE ROLE
CREATE DATABASE
$ echo 'Idempotent Database Schema Management is easy.'
I'm an idiot.
```

# "Solution": Choose during Automation. Sure.

Maybe one way would be if the deployment script could say "We're missing column X and Y" from comparing a description of the db to the actual one. If you deploy an older version, you still get the message but **choose** not to apply the change if it's destructive.

**HACKER NEWS,**
**Strategies for Database Success, 2nd Edition**

# Bootstrapping a Database

```
$ psql
postgres=# CREATE ROLE pgcon WITH PASSWORD't:<fTuz(vqg?YLY+pASY#}KO3(*&@6' LOGIN;
CREATE ROLE
postgres=# CREATE DATABASE pgcon OWNER pgcon;
CREATE DATABASE
postgres=# \c pgcon pgcon
You are now connected to database "pg" as user "pgcon".
pgcon=> \i pgcon-schema.sql
FATAL ERROR: Slippery Slope Encountered.  *PANIC*
```

# Doing It Right: Terraform Edition

# Bootstrapping an Understanding of Terraform

# Terraform: Input Globbing

- Provider and Resource Configuration data can exist in the same file
- Idioms:
  - Variables defined in: `interface.tf`
  - Providers defined in: `provider.tf`
  - One file per collection of related objects or resources (user discretion):
    E.g. `my-app.tf`, `my-servers.tf`, `my-service.tf`
- Personal Preferences:
  - Use a `GNUmakefile` to drive common operations: `make plan`, *THINK*, `make apply`
  - Ship a commented example `.vars` file: `cp .terraform.vars{.example,}`
  - "Hide" state or variables in dot files:
    E.g.: `.terraform.state`, `.terraform.plan`, `.terraform.vars`

# Terraform Inputs

provider
`*.tf`

config
*.tf

state file
`*.state`
`*.tfstate`

variables
`*.vars`
`*.tfvars`

```
terraform plan
terraform apply
terraform import
terraform show
```

# Provider

- Describes an endpoint that you work with
- Examples:
  - PostgreSQL endpoint: `postgresql://myuser@db1.internal:5432/mydb`
  - API URL: https://us-west-1.api.joyentcloud.com

# Provider Examples

```
$ cat provider.tf
provider "triton" {
  account       = "${var.triton_account_name}"
  #key_material = "${file(var.triton_key_material)}"
  key_id        = "${var.triton_key_id}"
  url           = "${var.triton_url}"
}

provider "postgresql" {
  alias           = "pg1"
  host            = "${triton_machine.postgresql96.primaryip}"
  port            = 5432
  database        = "pgcon"
  username        = "pgcon"
  password        = "t:<fTuz(vqg?YLY+pASY#}KO3(*&@6"
  sslmode         = "disable"
  connect_timeout = 15
}
```

Provider Name

# Provider Examples

```
$ cat provider.tf
provider "triton" {
  account       = "${var.triton_account_name}"
  #key_material = "${file(var.triton_key_material)}"
  key_id        = "${var.triton_key_id}"
  url           = "${var.triton_url}"
}

provider "postgresql" {
  alias           = "pg1"
  host            = "${triton_machine.postgresql96.primaryip}"
  port            = 5432
  database        = "pgcon"
  username        = "pgcon"
  password        = "t:<fTuz(vqg?YLY+pASY#}KO3(*&@6"
  sslmode         = "disable"
  connect_timeout = 15
}
```

Provider Attribute Names

# Provider Examples

```
$ cat provider.tf
provider "triton" {
  account       = "${var.triton_account_name}"
  #key_material = "${file(var.triton_key_material)}"
  key_id        = "${var.triton_key_id}"
  url           = "${var.triton_url}"
}

provider "postgresql" {
  alias           = "pg1"
  host            = "${triton_machine.postgresql96.primaryip}"
  port            = 5432
  database        = "pgcon"
  username        = "pgcon"
  password        = "t:<fTuz(vqg?YLY+pASY#}KO3(*&@6"
  sslmode         = "disable"
  connect_timeout = 15
}
```

Provider Attribute Values

# Provider Examples

```
$ cat provider.tf
provider "triton" {
  account       = "${var.triton_account_name}"
  #key_material = "${file(var.triton_key_material)}"
  key_id        = "${var.triton_key_id}"
  url           = "${var.triton_url}"
}

provider "postgresql" {
  alias           = "pg1"
  host            = "${triton_machine.postgresql96.primaryip}"
  port            = 5432
  database        = "pgcon"
  username        = "pgcon"
  password        = "t:<fTuz(vqg?YLY+pASY#}KO3(*&@6"
  sslmode         = "disable"
  connect_timeout = 15
}
```

Variable Interpolation

## Provider Examples

```
$ cat provider.tf
provider "triton" {
  account       = "${var.triton_account_name}"
  #key_material = "${file(var.triton_key_material)}"
  key_id        = "${var.triton_key_id}"
  url           = "${var.triton_url}"
}

provider "postgresql" {
  alias           = "pg1"
  host            = "${triton_machine.postgresql96.primaryip}"
  port            = 5432
  database        = "pgcon"
  username        = "pgcon"
  password        = "t:<fTuz(vqg?YLY+pASY#}KO3(*&@6"
  sslmode         = "disable"
  connect_timeout = 15
}
```

QUIRK ALERT: Variable interpolation creates a new scope. This is valid HCL:

```
key_material = "${file("~/.ssh/id_rsa")}"
```

Random demo PSA: don't use **`key_material`** in practice (it's here for explanation purposes only)

# Provider Examples

Helper Function: `file()`

```
$ cat provider.tf
provider "triton" {
  account       = "${var.triton_account_name}"
  #key_material = "${file(var.triton_key_material)}"
  key_id        = "${var.triton_key_id}"
  url           = "${var.triton_url}"
}

provider "postgresql" {
  alias           = "pg1"
  host            = "${triton_machine.postgresql96.primaryip}"
  port            = 5432
  database        = "pgcon"
  username        = "pgcon"
  password        = "t:<fTuz(vqg?YLY+pASY#}KO3(*&@6"
  sslmode         = "disable"
  connect_timeout = 15
}
```

Variable Interpolation, Out Parameter

# Provider Examples

Variable Interpolation, (User || File) Parameter

Name of User-Supplied Variable

```
$ cat provider.tf
provider "triton" {
  account       = "${var.triton_account_name}"
  #key_material = "${file(var.triton_key_material)}"
  key_id        = "${var.triton_key_id}"
  url           = "${var.triton_url}"
}

provider "postgresql" {
  alias           = "pg1"
  host            = "${triton_machine.postgresql96.primaryip}"
  port            = 5432
  database        = "pgcon"
  username        = "pgcon"
  password        = "t:<fTuz(vqg?YLY+pASY#}KO3(*&@6"
  sslmode         = "disable"
  connect_timeout = 15
}
```

# Provider Examples

```
$ cat provider.tf
provider "triton" {
  account       = "${var.triton_account_name}"
  #key_material = "${file(var.triton_key_material)}"
  key_id        = "${var.triton_key_id}"
  url           = "${var.triton_url}"
}

provider "postgresql" {
  alias           = "pg1"
  host            = "${triton_machine.postgresql96.primaryip}"
  port            = 5432
  database        = "pgcon"
  username        = "pgcon"
  password        = "t:<fTuz(vqg?YLY+pASY#}KO3(*&@6"
  sslmode         = "disable"
  connect_timeout = 15
}
```

Resource Type

Resource Out Parameter

Resource Name

# Config Files

- Declarative language specifying what you want the world to look like
- Supports variables
- Supports config file reuse via modules (not covered in this talk)
- Should be version controlled

# Config File

```
resource "postgresql_database" "db1" {
  provider          = "postgresql.pg1"
  name              = "pgcon"
  owner             = "pgcon"
  template          = "template0"
  lc_collate        = "C"
  connection_limit  = -1
  allow_connections = true
}

resource "postgresql_sequence" "table1_seq" {
  name = "table1_seq"
}

resource "postgresql_column" "table1_id" {
  name     = "id"
  type     = "INT8"
  not_null = true
  default  = "nextval(${postgresql_sequence.table1_seq.regclass})"
}

resource "postgresql_table" "table1" {
  name    = "table1"
  schema  = "${postgresql_schema.foo_service.name}"
  columns = [ "${postgresql_column.table1_id.id}" ]
}
```

Resource Type

# Config File

```
resource "postgresql_database" "db1" {
  provider          = "postgresql.pg1"
  name              = "pgcon"
  owner             = "pgcon"
  template          = "template0"
  lc_collate        = "C"
  connection_limit  = -1
  allow_connections = true
}

resource "postgresql_sequence" "table1_seq" {
  name = "table1_seq"
}

resource "postgresql_column" "table1_id" {
  name     = "id"
  type     = "INT8"
  not_null = true
  default  = "nextval(${postgresql_sequence.table1_seq.regclass})"
}

resource "postgresql_table" "table1" {
  name    = "table1"
  schema  = "${postgresql_schema.foo_service.name}"
  columns = [ "${postgresql_column.table1_id.id}" ]
}
```

Resource Name

# Terraform is like an Onion: It Has Layers

- One provider per connection DSN
- Different resources can pull from different providers
  - Specifying a provider per resource is clunky and not worth the effort
  - Doable if the number of resources per provider is relatively small
  - You're all in on provider aliases or you're not
- Create two providers, each in a different directory:
  1. PostgreSQL superuser provider: connects to postgres DB as a superuser
  2. PostgreSQL per-database administrator: connects to the respective DB as DB owner
- Create the necessary config in each directory for each database

# Bootstrapping a Database

```
~seanc/src/pgcon2017 $ cd superuser
~seanc/src/pgcon2017/superuser $ make plan
terraform plan -state=.terraform.state -var-file=.terraform.vars -out=.terraform.plan
Refreshing Terraform state in-memory prior to plan...
[snip]
+ postgresql_database.db1
    allow_connections: "true"
    connection_limit:  "-1"
    encoding:          "<computed>"
    is_template:       "<computed>"
    lc_collate:        "C"
    lc_ctype:          "<computed>"
    name:              "pgcon"
    owner:             "pgcon"
    tablespace_name:   "<computed>"
    template:          "template0"

+ postgresql_role.pgcon
    bypass_row_level_security: "false"
    connection_limit:          "5"
    create_database:           "false"
    create_role:               "false"
    encrypted_password:        "true"
    inherit:                   "true"
    login:                     "true"
    name:                      "pgcon"
    password:                  "<sensitive>"
    replication:               "false"
    skip_drop_role:            "false"
    skip_reassign_owned:       "false"
    superuser:                 "false"
    valid_until:               "infinity"


Plan: 2 to add, 0 to change, 0 to destroy.
```

# Bootstrapping a Database

```
% make apply
terraform apply -state-out=.terraform.state .terraform.plan
postgresql_role.pgcon: Creating...
  bypass_row_level_security: "" => "false"
  connection_limit:          "" => "5"
  create_database:           "" => "false"
  create_role:               "" => "false"
  encrypted_password:        "" => "true"
  inherit:                   "" => "true"
  login:                     "" => "true"
  name:                      "" => "pgcon"
  password:                  "<sensitive>" => "<sensitive>"
  replication:               "" => "false"
  skip_drop_role:            "" => "false"
  skip_reassign_owned:       "" => "false"
  superuser:                 "" => "false"
  valid_until:               "" => "infinity"
postgresql_role.pgcon: Creation complete (ID: pgcon)
```

```
postgresql_database.db1: Creating...
  allow_connections: "" => "true"
  connection_limit:  "" => "-1"
  encoding:          "" => "<computed>"
  is_template:       "" => "<computed>"
  lc_collate:        "" => "C"
  lc_ctype:          "" => "<computed>"
  name:              "" => "pgcon"
  owner:             "" => "pgcon"
  tablespace_name:   "" => "<computed>"
  template:          "" => "template0"
postgresql_database.db1: Creation complete (ID: pgcon)

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

The state of your infrastructure has been saved to the path
below. This state is required to modify and destroy your
infrastructure, so keep it safe. To inspect the complete state
use the `terraform show` command.

State path: .terraform.state
```

# Bootstrapping a Database

```
~seanc/src/pgcon2017/superuser $ make plan
terraform plan -state=.terraform.state -var-file=.terraform.vars -out=.terraform.plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

postgresql_role.pgcon: Refreshing state... (ID: pgcon)
postgresql_database.db1: Refreshing state... (ID: pgcon)
No changes. Infrastructure is up-to-date.

This means that Terraform did not detect any differences between your
configuration and real physical resources that exist. As a result, Terraform
doesn't need to do anything.
```

# Bootstrapping a Database

```
~seanc/src/pgcon2017/superuser $ cat provider.tf
provider "postgresql" {
  # alias          = "pg-db1"
  host            = "127.0.0.1"
  port            = 5432
  database        = "postgres"
  username        = "postgres"
  password        = "t:<fTuz(vqg?YLY+pASY#}KO3(*&@6"
  sslmode        = "disable"
  connect_timeout = 15
}
```

# Bootstrapping a Database

```
~seanc/src/pgcon2017/superuser $ make psql
postgres@[local]:5432/postgres# \l
                                List of databases
   Name    |  Owner   | Encoding | Collate | Ctype |   Access privileges
-----------+----------+----------+---------+-------+-----------------------
 pgcon     | pgcon    | UTF8     | C       | C     |
 postgres  | postgres | UTF8     | C       | C     |
 template0 | postgres | UTF8     | C       | C     | =c/postgres          +
           |          |          |         |       | postgres=CTc/postgres
 template1 | postgres | UTF8     | C       | C     | =c/postgres          +
           |          |          |         |       | postgres=CTc/postgres
(4 rows)
postgres@[local]:5432/postgres# \du
                                    List of roles
 Role name |                          Attributes                          | Member of
-----------+--------------------------------------------------------------+-----------
 pgcon     | 5 connections                                               +| {}
           | Password valid until infinity                                |
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {pgcon}
postgres@[local]:5432/postgres# DROP DATABASE pgcon;
DROP DATABASE
Time: 247.785 ms
```

# Bootstrapping a Database

```
~seanc/src/pgcon2017/superuser $ make plan
terraform plan -state=.terraform.state -var-file=.terraform.vars -out=.terraform.plan
Refreshing Terraform state in-memory prior to plan...
[snip]
+ postgresql_database.db1
    allow_connections:  "true"
    connection_limit:   "-1"
    encoding:           "<computed>"
    is_template:        "<computed>"
    lc_collate:         "C"
    lc_ctype:           "<computed>"
    name:               "pgcon"
    owner:              "pgcon"
    tablespace_name:    "<computed>"
    template:           "template0"


Plan: 1 to add, 0 to change, 0 to destroy.
```

```
~seanc/src/pgcon2017/superuser $ make apply
terraform apply -state-out=.terraform.state .terraform.plan
postgresql_database.db1: Creating...
  allow_connections:  "" => "true"
  connection_limit:   "" => "-1"
  encoding:           "" => "<computed>"
  is_template:        "" => "<computed>"
  lc_collate:         "" => "C"
  lc_ctype:           "" => "<computed>"
  name:               "" => "pgcon"
  owner:              "" => "pgcon"
  tablespace_name:    "" => "<computed>"
  template:           "" => "template0"
postgresql_database.db1: Creation complete (ID: pgcon)

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

The state of your infrastructure has been saved to the path
below. This state is required to modify and destroy your
infrastructure, so keep it safe. To inspect the complete state
use the `terraform show` command.

State path: .terraform.state
```

# Bootstrapping a Database: Moral Convictions

```
~seanc/src/pgcon2017/superuser $ make cat-logs | faux-grep ${SOME_INTERESTING_STUFF} | tail -n 1-ish
LOG:  connection authorized: user=postgres database=postgres
LOG:  statement: GRANT "pgcon" TO "postgres"
LOG:  statement: CREATE DATABASE "pgcon" OWNER "pgcon" TEMPLATE "template0" ENCODING "UTF8" LC_COLLATE "C" LC_CTYPE "C" ALLOW_CONNECTIONS
true CONNECTION LIMIT -1 IS_TEMPLATE false
LOG:  execute <unnamed>: SELECT d.datname, pg_catalog.pg_get_userbyid(d.datdba) from pg_database d WHERE datname=$1
DETAIL:  parameters: $1 = 'pgcon'
LOG:  statement: REVOKE "pgcon" FROM "postgres"
```

We can do better than that

# Bootstrapping a Database: Delegated Privileges

```
~seanc/src/pgcon2017/superuser $ make psql
postgres@[local]:5432/postgres# CREATE ROLE tf_dba CONNECTION LIMIT 5 LOGIN CREATEROLE CREATEDB;
CREATE ROLE
postgres@[local]:5432/postgres# DROP DATABAZSE pgcon;
DROP DATABASE
~seanc/src/pgcon2017/superuser $ cat provider.tf
provider "postgresql" {
  host            = "127.0.0.1"
  port            = 5432
  database        = "postgres"
  username        = "tf_dba"
  password        = "t:<fTuz(vqg?YLY+pASY#}KO3(*&@6"
  sslmode        = "disable"
  connect_timeout = 15
}
~seanc/src/pgcon2017/superuser $ make plan apply
/Users/seanc/go/bin/terraform plan -state=.terraform.state -var-file=.terraform.vars -out=.terraform.plan
Refreshing Terraform state in-memory prior to plan...
postgresql_role.pgcon: Refreshing state... (ID: pgcon)
postgresql_database.db1: Refreshing state... (ID: pgcon)
+ postgresql_database.db1
Plan: 1 to add, 0 to change, 0 to destroy.
/Users/seanc/go/bin/terraform apply -state-out=.terraform.state .terraform.plan
postgresql_database.db1: Creating...
postgresql_database.db1: Creation complete (ID: pgcon)

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

# Bootstrapping a Database: Trust, but Verify

```
~seanc/src/pgcon2017/superuser $ make cat-logs | faux-grep ${SOME_INTERESTING_STUFF} | tail -n 1-ish
LOG:  connection received: host=127.0.0.1 port=50247
LOG:  connection authorized: user=tf_dba database=postgres
LOG:  statement: GRANT "pgcon" TO "tf_dba"
LOG:  statement: CREATE DATABASE "pgcon" OWNER "pgcon" TEMPLATE "template0" ENCODING "UTF8" LC_COLLATE "C" LC_CTYPE "C" ALLOW_CONNECTIONS
true CONNECTION LIMIT -1 IS_TEMPLATE false
LOG:  execute <unnamed>: SELECT d.datname, pg_catalog.pg_get_userbyid(d.datdba) from pg_database d WHERE datname=$1
DETAIL:  parameters: $1 = 'pgcon'
LOG:  statement: REVOKE "pgcon" FROM "tf_dba"
```

Excellent

# Drift Detection: Periodic Audits

```
~seanc/src/pgcon2017/superuser $ make plan
terraform plan -state=.terraform.state -var-file=.terraform.vars -out=.terraform.plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

postgresql_role.pgcon: Refreshing state... (ID: pgcon)
postgresql_database.db1: Refreshing state... (ID: pgcon)
No changes. Infrastructure is up-to-date.

This means that Terraform did not detect any differences between your
configuration and real physical resources that exist. As a result, Terraform
doesn't need to do anything.
```

Good

# Drift Detection: Intruder Alert

```
~seanc/src/pgcon2017/superuser $ make psql
postgres@[local]:5432/postgres# ALTER DATABASE pgcon OWNER TO tf_dba;
ALTER DATABASE
~seanc/src/pgcon2017/superuser $ make plan
terraform plan -state=.terraform.state -var-file=.terraform.vars -out=.terraform.plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

postgresql_role.pgcon: Refreshing state... (ID: pgcon)
postgresql_database.db1: Refreshing state... (ID: pgcon)
The Terraform execution plan has been generated and is shown below.
Resources are shown in alphabetical order for quick scanning. Green resources
will be created (or destroyed and then created if an existing resource
exists), yellow resources are being changed in-place, and red resources
will be destroyed. Cyan entries are data sources to be read.

Your plan was also saved to the path below. Call the "apply" subcommand
with this plan file and Terraform will exactly execute this execution
plan.

Path: .terraform.plan

~ postgresql_database.db1
    owner: "tf_dba" => "pgcon"


Plan: 0 to add, 1 to change, 0 to destroy.
```

Evil always wins because good is dumb. -Lord Helmet

Seek and Destroy Alternate Facts

# Drift Detection: Uprising Quelled

```
~seanc/src/pgcon2017/superuser $ make apply
terraform apply -state-out=.terraform.state .terraform.plan
postgresql_database.db1: Modifying... (ID: pgcon)
  owner: "tf_dba" => "pgcon"
postgresql_database.db1: Modifications complete (ID: pgcon)

Apply complete! Resources: 0 added, 1 changed, 0 destroyed.

The state of your infrastructure has been saved to the path
below. This state is required to modify and destroy your
infrastructure, so keep it safe. To inspect the complete state
use the `terraform show` command.

State path: .terraform.state
```

Good always wins because evil is dumb (or less diligent).
-me

# Superuser Inputs: Connection Truth

```
~seanc/src/pgcon2017/superuser $ ls -lA
-rw-r--r--  1 seanc  staff  4509 May 24 15:09 .terraform.plan
-rw-r--r--  1 seanc  staff  2681 May 24 14:59 .terraform.state
-rw-r--r--  1 seanc  staff  2681 May 24 14:59 .terraform.state.backup
lrwxr-xr-x  1 seanc  staff    18 May 24 12:58 .terraform.vars@ -> ../.terraform.vars
lrwxr-xr-x  1 seanc  staff    26 May 24 12:58 .terraform.vars.example@ -> ../.terraform.vars.example
lrwxr-xr-x  1 seanc  staff    14 May 24 12:58 GNUmakefile@ -> ../GNUmakefile
-rw-r--r--  1 seanc  staff   520 May 24 13:34 db-pgcon.tf
-rw-------  1 seanc  staff   278 May 24 14:29 provider.tf
~seanc/src/pgcon2017/superuser $ cat provider.tf
variable "tf_dba_username" {
  type    = "string"
  default = "tf_dba"
}

variable "tf_dba_password" {
  type = "string"
}

provider "postgresql" {
  # alias         = "pg-db1"
  host            = "127.0.0.1"
  port            = 5432
  database        = "postgres"
  username        = "${var.tf_dba_username}"
  password        = "${var.tf_dba_password}"
  sslmode         = "disable"
  connect_timeout = 15
}
```

# Superuser Inputs: Database Metadata Truth

```
~seanc/src/pgcon2017/superuser $ cat GNUmakefile
See https://github.com/sean-/joyent-freebsd
~seanc/src/pgcon2017/superuser $ ls -lA
total 72
-rw-r--r--  1 seanc  staff  4509 May 24 15:09 .terraform.plan
-rw-r--r--  1 seanc  staff  2681 May 24 14:59 .terraform.state
-rw-r--r--  1 seanc  staff  2681 May 24 14:59 .terraform.state.backup
lrwxr-xr-x  1 seanc  staff    18 May 24 12:58 .terraform.vars@ -> ../.terraform.vars
lrwxr-xr-x  1 seanc  staff    14 May 24 12:58 GNUmakefile@ -> ../GNUmakefile
-rw-r--r--  1 seanc  staff   520 May 24 13:34 db-pgcon.tf
-rw-------  1 seanc  staff   278 May 24 14:29 provider.tf
~seanc/src/pgcon2017/superuser $ cat db-pgcon.tf
resource "postgresql_role" "pgcon" {
  # provider        = "postgresql.pg1"
  name              = "pgcon"
  replication       = false
  login             = true
  connection_limit = 5
  password          = "md5cc4cfa7bb9d7813887fa3526c468e5b0"
}

resource "postgresql_database" "db1" {
  # provider         = "postgresql.pg1"
  name               = "pgcon"
  owner              = "${postgresql_role.pgcon.name}"
  template           = "template0"
  lc_collate         = "C"
  connection_limit   = -1
  allow_connections = true
}
```

# What if I have a database already?

1. Write the connection information into a provider config block
2. Write the `postgresql_database` resource config block
3. Import the existing config into a state file using `terraform import`

# Superuser Inputs: Database Metadata Truth

```
~seanc/src/pgcon2017/superuser $ rm -f .terraform.state*
~seanc/src/pgcon2017/superuser $ make import PROVIDER=postgresql RESOURCE_ID=postgresql_role.pgcon LOOKUP_ID=pgcon
terraform import -state=.terraform.state -state-out=.terraform.state -var-file=.terraform.vars \
                 -provider=postgresql postgresql_role.pgcon pgcon
postgresql_role.pgcon: Importing from ID "pgcon"...
postgresql_role.pgcon: Import complete!
  Imported postgresql_role (ID: pgcon)
postgresql_role.pgcon: Refreshing state... (ID: pgcon)

Import success! The resources imported are shown above. These are
now in your Terraform state. Import does not currently generate
configuration, so you must do this next. If you do not create configuration
for the above resources, then the next `terraform plan` will mark
them for destruction.
~seanc/src/pgcon2017/superuser $ make show
terraform show .terraform.state
postgresql_role.pgcon:
  id = pgcon
  bypass_row_level_security = false
  connection_limit = 5
  create_database = false
  create_role = false
  encrypted_password = true
  inherit = true
  login = true
  name = pgcon
  password = md5cc4cfa7bb9d7813887fa3526c468e5b0
  replication = false
  skip_drop_role = false
  skip_reassign_owned = false
```

# Superuser Inputs: Database Metadata Truth

```
~seanc/src/pgcon2017/superuser $ make import PROVIDER=postgresql RESOURCE_ID=postgresql_database.db1 LOOKUP_ID=pgcon
terraform import -state=.terraform.state -state-out=.terraform.state -var-file=.terraform.vars \
                 -provider=postgresql postgresql_database.db1 pgcon
postgresql_database.db1: Importing from ID "pgcon"...
postgresql_database.db1: Import complete!
  Imported postgresql_database (ID: pgcon)
postgresql_database.db1: Refreshing state... (ID: pgcon)

Import success! The resources imported are shown above. These are
now in your Terraform state. Import does not currently generate
configuration, so you must do this next. If you do not create configuration
for the above resources, then the next `terraform plan` will mark
them for destruction.
~seanc/src/pgcon2017/superuser $ make show
terraform show .terraform.state
postgresql_database.db1:
  id = pgcon
  allow_connections = true
  connection_limit = -1
  encoding = UTF8
  is_template = false
  lc_collate = C
  lc_ctype = C
  name = pgcon
  owner = pgcon
  tablespace_name = pg_default
  template = template0
postgresql_role.pgcon:
  id = pgcon
  bypass_row_level_security = false
```

# Bootstrapping a Database

```
~seanc/src/pgcon2017/superuser $ cd ../db-pgcon
~seanc/src/pgcon2017/db-pgcon $ make plan
terraform plan -state=.terraform.state -var-file=.terraform.vars -out=.terraform.plan
Refreshing Terraform state in-memory prior to plan...
[snip]
+ postgresql_database.db1
    allow_connections:  "true"
    connection_limit:   "-1"
    encoding:           "<computed>"
    is_template:        "<computed>"
    lc_collate:         "C"
    lc_ctype:           "<computed>"
    name:               "pgcon"
    owner:              "pgcon"
    tablespace_name:    "<computed>"
    template:           "template0"

+ postgresql_role.pgcon
    bypass_row_level_security: "false"
    connection_limit:          "5"
    create_database:           "false"
    create_role:               "false"
    encrypted_password:        "true"
    inherit:                   "true"
    login:                     "true"
    name:                      "pgcon"
    password:                  "<sensitive>"
    replication:               "false"
    skip_drop_role:            "false"
    skip_reassign_owned:       "false"
    superuser:                 "false"
    valid_until:               "infinity"


Plan: 2 to add, 0 to change, 0 to destroy.
```

# Config File

```
$ cat my-users.tf
resource "postgresql_role" "pgcon" {
  provider         = "postgresql.pg1"
  name             = "pgcon"
  replication      = false
  login            = true
  connection_limit = 5
  password         = "md5cc4cfa7bb9d7813887fa3526c468e5b0"
}
$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

triton_machine.postgresql96: Refreshing state... (ID: e84e154b-ca39-639a-e9c1-a00c52079713)
postgresql_database.db1: Refreshing state... (ID: pgcon)
The Terraform execution plan has been generated and is shown below.
Resources are shown in alphabetical order for quick scanning. Green resources
will be created (or destroyed and then created if an existing resource
exists), yellow resources are being changed in-place, and red resources
will be destroyed. Cyan entries are data sources to be read.
```

# Config File

```
$ terraform plan
[snip]
Your plan was also saved to the path below. Call the "apply" subcommand
with this plan file and Terraform will exactly execute this execution
plan.

Path: .terraform.plan

+ postgresql_role.pgcon
    bypass_row_level_security: "false"
    connection_limit:          "5"
    create_database:           "false"
    create_role:               "false"
    encrypted_password:        "true"
    inherit:                   "true"
    login:                     "true"
    name:                      "pgcon"
    password:                  "<sensitive>"
    replication:               "false"
    skip_drop_role:            "false"
    skip_reassign_owned:       "false"
    superuser:                 "false"
    valid_until:               "infinity"


Plan: 2 to add, 0 to change, 0 to destroy.
```

## Variables

- Passed in via environment variables
- Must be pre-declared in terraform configuration files
- CLI flag: **`-var 'foo=bar'`**
- Variables file: **`-var-file=.terraform.vars`**
- Missing variables prompt for user input
- "Out parameters" from previously evaluated vertices in the dependency graph

```
some_parameter = "${var.foo}"
host           = "${triton_machine.postgresql96.primaryip}"
```

# `interface.tf`: Input Variables

- `interface.tf` is a convention, not a requirement. `vars.tf` was vogue for a while but misleading.
- `interface.tf` has three sections:
  1. Input variables:

     ```
     variable "my_variable_name" {
       # map, list, or string are the three supported types
       type        = "string" # string is the default, can be omitted
       default     = "default_value" # Optional
       description = "explain this variable"
     }
     ```

  2. Modules (think dependencies)
  3. Output variables

# `interface.tf`: Modules

- **`interface.tf`** is a convention, not a requirement. `vars.tf` was vogue for a while but misleading.
- **`interface.tf`** has three sections:
  1. Input variables
  2. Modules (think dependencies):

     ```
     module "my_local_name" {
       source = "../path/to/module/"
     }
     # "${my_local_name.an_output_variable}"
     ```
  3. Output variables

# `interface.tf`: Output Variables

- **`interface.tf`** is a convention, not a requirement. `vars.tf` was vogue for a while but misleading.
- **`interface.tf`** has three sections:
    1. Input variables
    2. Modules (think dependencies)
    3. Output variables (a.k.a. "out parameters"):

```
output "an_output_variable" {
  value = "${postgresql_database.db1.name}"
}
```

# Variables File

```
$ cat interface.tf
variable "pgcon_username" {
  type    = "string"
  default = "pgcon"
}

variable "pgcon_password" {
  type = "string"
}

module "pgcon_database" {
  source = "../superuser"
}

output "dbname" {
  variable = "${postgresql_database.db1.name}"
}

output "owner" {
  variable = "${postgresql_database.db1.owner}"
}
```

# Variables File

```
$ cat .terraform.vars
#db_network_id = "01234567-89ab-cdef-0123-456789abcdef"

# Image ID from the output of the packer run, or `make my-images`
#postgresql96_image_id = "deadbeef-c0ff-eede-adbe-efc0ffeebeef"

# Package name from `make packages`
#my_image_size = "k4-highcpu-kvm-250M"

# Triton account information. Example:
#
# $ make env | grep ^SDC_ACCOUNT | cut -d '=' -f2
#triton_account_name = "my-triton-account-name"

# Key ID from `triton keys` that matches the file used in key_material. Example:
#
# $ make env | grep ^SDC_KEY_ID | cut -d '=' -f2
#triton_key_id = "de:ad:be:ef:c0:ff:ee:de:ad:be:ef:c0:ff:ee:be:ef"

# Path to your SSH key
#triton_key_material = "~/.ssh/id_rsa"

# URL from `triton datacenters`.  Example:
#
# $ make env | grep ^SDC_URL | cut -d '=' -f2
#triton_url = "https://us-west-1.api.joyentcloud.com"
```

# Automatic Dependencies

```
resource "postgresql_database" "db1" {
 name                  = "pgcon"
  owner                 = "${postgresql_role.pgcon.name}"
  template              = "template0"
  lc_collate            = "C"
  connection_limit  = -1
  allow_connections = true
}

resource "postgresql_role" "pgcon" {
 name                 = "${var.pgcon_username}"
  replication       = false
  login             = true
  connection_limit = 5
  password          = "${var.pgcon_password}"
}
```

1) Variable Reference

2) Creates DAG edge

# Automatic Dependencies

```
resource "postgresql_database" "db1" {
 name                  = "pgcon"
  owner                 = "${postgresql_role.pgcon.name}"
  template              = "template0"
  lc_collate            = "C"
  connection_limit      = -1
  allow_connections = true
}

resource "postgresql_role" "pgcon" {
 name                 = "${var.pgcon_username}"
  replication          = false
  login                = true
  connection_limit = 5
  password             = "${var.pgcon_password}"
}
```

Evaluation Order:

1) `postgresql_role.pgcon`
2) `postgresql_database.db1`

Depth-first search of dependency graph

# State File

- Stateful snapshot of what Terraform thinks the world should be like
- Local to your Terraform process
- A big ball of JSON (feel free to look through it)
- TIP: Keep backups of your state file
- TIP: Don't allow concurrent manipulation of the state file

# State File

```
$ cat .terraform.state
{
    "version": 3,
    "terraform_version": "0.9.6",
    "serial": 12,
    "lineage": "cde58dde-8d9c-4f1d-b9dc-5e3660e70982",
    "modules": [
        {
            "path": [
                "root"
            ],
            "outputs": {},
            "resources": {
                "postgresql_database.db1": {
                    "type": "postgresql_database",
                    "depends_on": [],
                    "primary": {
                        "id": "pgcon",
                        "attributes": {
                            "allow_connections": "true",
                            "connection_limit": "-1",
                            "encoding": "UTF8",
                            "id": "pgcon",
                            "is_template": "false",
                            "lc_collate": "C",
```

# Importing a Database

# Import Procedure

1. Import a resource into your statefile
2. Write config until the plan converges

# Inevitabilities from Importing

1. Import a resource into your statefile
2. Write config until the plan converges

- You will fumble a resource name
- You will fumble a resource ID

PSA: Your plan will not converge until you fix the bug from import.

# Database Import

```
$ cat provider.tf
provider "postgresql" {
  alias           = "pg1"
  host            = "${triton_machine.postgresql96.primaryip}"
  port            = 5432
  database        = "pgcon"
  username        = "pgcon"
  password        = "t:<fTuz(vqg?YLY+pASY#}KO3(*&@6"
  sslmode         = "disable"
  connect_timeout = 15
}
```

# Database Import

```
$ cat provider.tf
provider "postgresql" {
  alias           = "pg1"
  host            = "${triton_machine.postgresql96.primaryip}"
  port            = 5432
  database        = "pgcon"
  username        = "pgcon"
  password        = "t:<fTuz(vqg?YLY+pASY#}KO3(*&@6"
  sslmode         = "disable"
  connect_timeout = 15
}
$ terraform import -state=.terraform.state -state-out=.terraform.state -var-file=.terraform.vars \
  -provider=postgresql.pg1 postgresql_database.db1 pgcon
postgresql_database.db1: Importing from ID "pgcon"...
postgresql_database.db1: Import complete!
  Imported postgresql_database (ID: pgcon)
postgresql_database.db1: Refreshing state... (ID: pgcon)

Import success! The resources imported are shown above. These are
now in your Terraform state. Import does not currently generate
configuration, so you must do this next. If you do not create configuration
for the above resources, then the next `terraform plan` will mark
them for destruction.
```

# Database Import

```
$ terraform import -state=.terraform.state -state-out=.terraform.state -var-file=.terraform.vars \
  -provider=postgresql.pg1 postgresql_database.db1 pgcon
postgresql_database.db1: Importing from ID "pgcon"...
postgresql_database.db1: Import complete!
  Imported postgresql_database (ID: pgcon)
postgresql_database.db1: Refreshing state... (ID: pgcon)

Import success! The resources imported are shown above. These are
now in your Terraform state. Import does not currently generate
configuration, so you must do this next. If you do not create configuration
for the above resources, then the next `terraform plan` will mark
them for destruction.
$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

triton_machine.postgresql96: Refreshing state... (ID: e84e154b-ca39-639a-e9c1-a00c52079713)
postgresql_database.db1: Refreshing state... (ID: pgcon)
No changes. Infrastructure is up-to-date.

This means that Terraform did not detect any differences between your
configuration and real physical resources that exist. As a result, Terraform
doesn't need to do anything.
```

# Duplicate Imports

```
$ terraform import -state=.terraform.state -state-out=.terraform.state -var-file=.terraform.vars \
  -provider=postgresql.pg1 postgresql_role.pgcon pgcon
postgresql_role.pgcon: Importing from ID "pgcon"...
postgresql_role.pgcon: Import complete!
  Imported postgresql_role (ID: pgcon)
Error importing: 1 error(s) occurred:

* postgresql_role.db1 (import id: pgcon): Can't import postgresql_role.pgcon, would collide with an existing resource.

Please remove or rename this resource before continuing.
```

Incorrect Resource Name

# Database Import

```
$ cat my-users.tf
resource "postgresql_role" "pgcon" {
  provider         = "postgresql.pg1"
  name             = "pgcon"
  replication      = false
  login            = true
  connection_limit = 5
  password         = "md5cc4cfa7bb9d7813887fa3526c468e5b0"
}
$ terraform import -state=.terraform.state -state-out=.terraform.state -var-file=.terraform.vars \
  -provider=postgresql.pg1 postgresql_role.db1 pgcon
postgresql_role.db1: Importing from ID "pgcon"...
postgresql_role.db1: Import complete!
  Imported postgresql_role (ID: pgcon)
postgresql_role.db1: Refreshing state... (ID: pgcon)

Import success! The resources imported are shown above. These are
now in your Terraform state. Import does not currently generate
configuration, so you must do this next. If you do not create configuration
for the above resources, then the next `terraform plan` will mark
them for destruction.
```

# Database Import

```
$ terraform import -state=.terraform.state -state-out=.terraform.state -var-file=.terraform.vars \
  -provider=postgresql.pg1 postgresql_role.db1 pgcon
postgresql_role.db1: Importing from ID "pgcon"...
postgresql_role.db1: Import complete!
  Imported postgresql_role (ID: pgcon)
postgresql_role.db1: Refreshing state... (ID: pgcon)

Import success! The resources imported are shown above. These are
now in your Terraform state. Import does not currently generate
configuration, so you must do this next. If you do not create configuration
for the above resources, then the next `terraform plan` will mark
them for destruction.
```

# Beginnings of a DAG

[root] root

[root] meta.count-boundary (count boundary fixup)

[root] output.pg_db1_name

[root] provider.postgresql (close)

[root] output.pg_db1_host

postgresql_database.db1

[root] output.pg_db1_port

postgresql_role.pgcon

[root] var.pgcon_username

[root] var.pgcon_password

provider.postgresql

[root] var.pg_host

[root] var.pg_port

[root] var.tf_dba_password

[root] var.tf_dba_username

# Iterative DB Design

# Bottom Up Schema Design

- Terraform can be used with string inputs
- Prefer variable interpolation instead

# Resource: `postgresql_role`

```
$ cat roles.tf
resource "postgresql_role" "app_www" {
  name             = "app_www"
  login            = true
  connection_limit = 100
}

resource "postgresql_role" "app_releng" {
  name             = "app_releng"
  login            = true
  connection_limit = 1
}

resource "postgresql_role" "role_with_defaults" {
  name                       = "testing_role_with_defaults"
  #superuser                 = false
  #create_database           = false
  #create_role               = false
  #inherit                   = false
  #login                     = false
  #replication               = false
  #bypass_row_level_security = false
  #connection_limit          = -1
  #encrypted_password        = true
  #password                  = ""
  #skip_drop_role            = false
  #skip_reassign_owned       = false
  #valid_until               = "infinity"
}
```

# Resource: `postgresql_schema`

```
$ cat schemas.tf
resource "postgresql_schema" "foo_service" {
  name  = "foo_service"
  owner = "${postgresql_role.app_releng.name}"

  policy {
    usage = true
    role  = "${postgresql_role.app_www.name}"
  }

  # app_releng can create new objects in the schema.  This is the role that
  # migrations are executed as.
  policy {
    create = true
    usage  = true
    role   = "${postgresql_role.app_releng.name}"
  }

  # app_dba can create new objects in the schema and GRANT. Anti-pattern, danger.
  policy {
    create_with_grant = true
    usage_with_grant  = true
    role              = "${postgresql_role.app_dba.name}"
  }
}
```

# Resource: `postgresql_sequence`

```
$ cat sequences.tf
resource "postgresql_sequence" "table1_seq" {
  name = "table1_seq"

  # temporary = false
  # increment_by = 1
  # min = 1
  # max = 2^63 - 1
  # start = 1
  # cache = 1
  # cycle = false
}
```

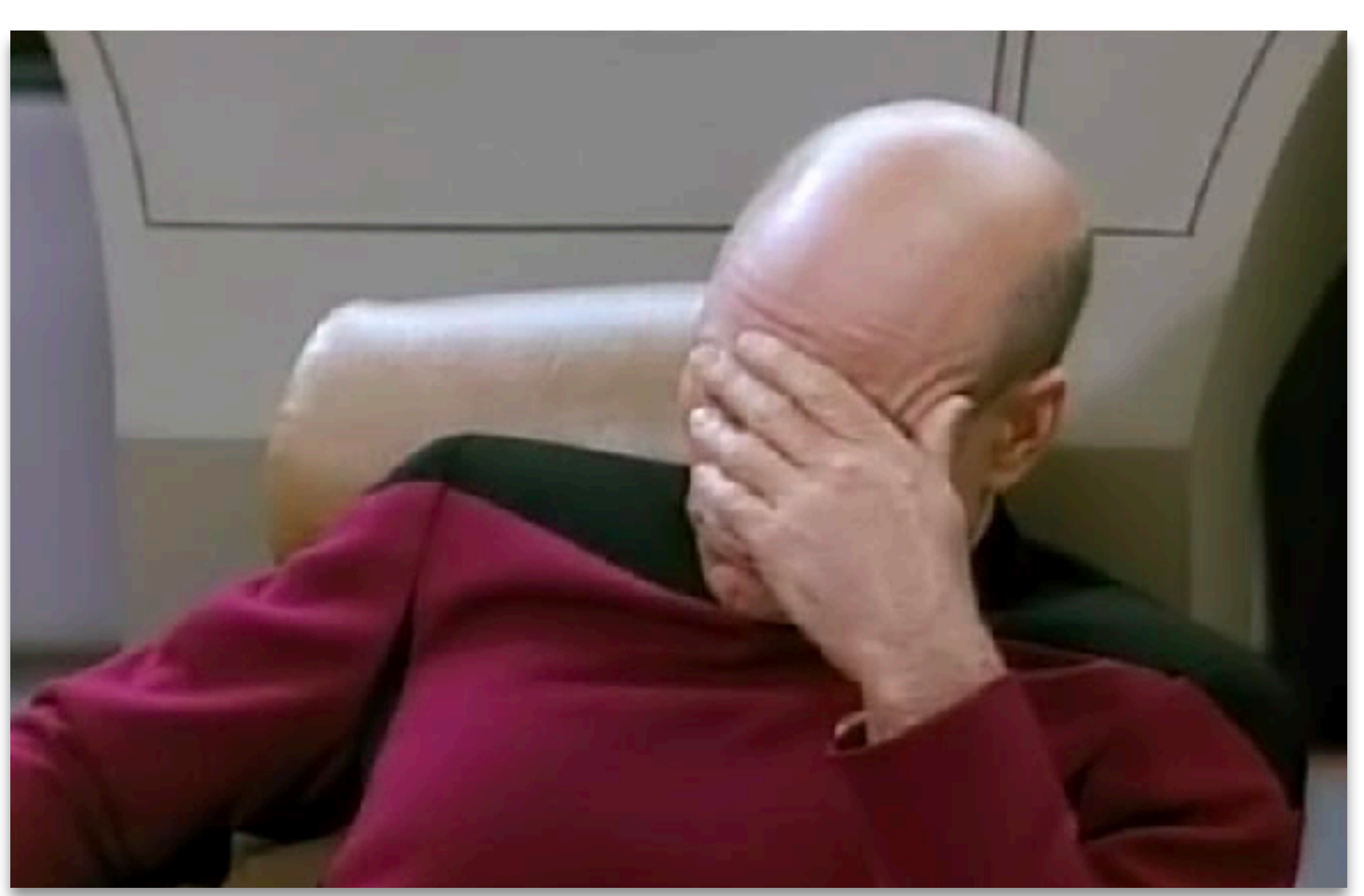
# postgresql_column and postgresql_table

```
$ cat tables.tf
resource "postgresql_table" "table1" {
  # provider = "postgresql.pg1"
  name   = "table1"
  schema = "${postgresql_schema.foo_service.name}"

  columns = [
    "${postgresql_column.table1_id.id}",
    "${postgresql_column.table1_username.id}",
  ]
}

resource "postgresql_column" "table1_id" {
  name     = "id"
  type     = "INT8"
  not_null = true
  default  = "nextval(${postgresql_sequence.table1_seq.regclass})"
}

resource "postgresql_column" "table1_username" {
  name     = "username"
  type     = "CITEXT"
  not_null = true
}
```

# Database Schemas

> "Do not perform table alter commands. Table alter commands introduce locks and downtimes."

HACKER_NEWS = ${SOMEONE_ON_THE_INTERNET}

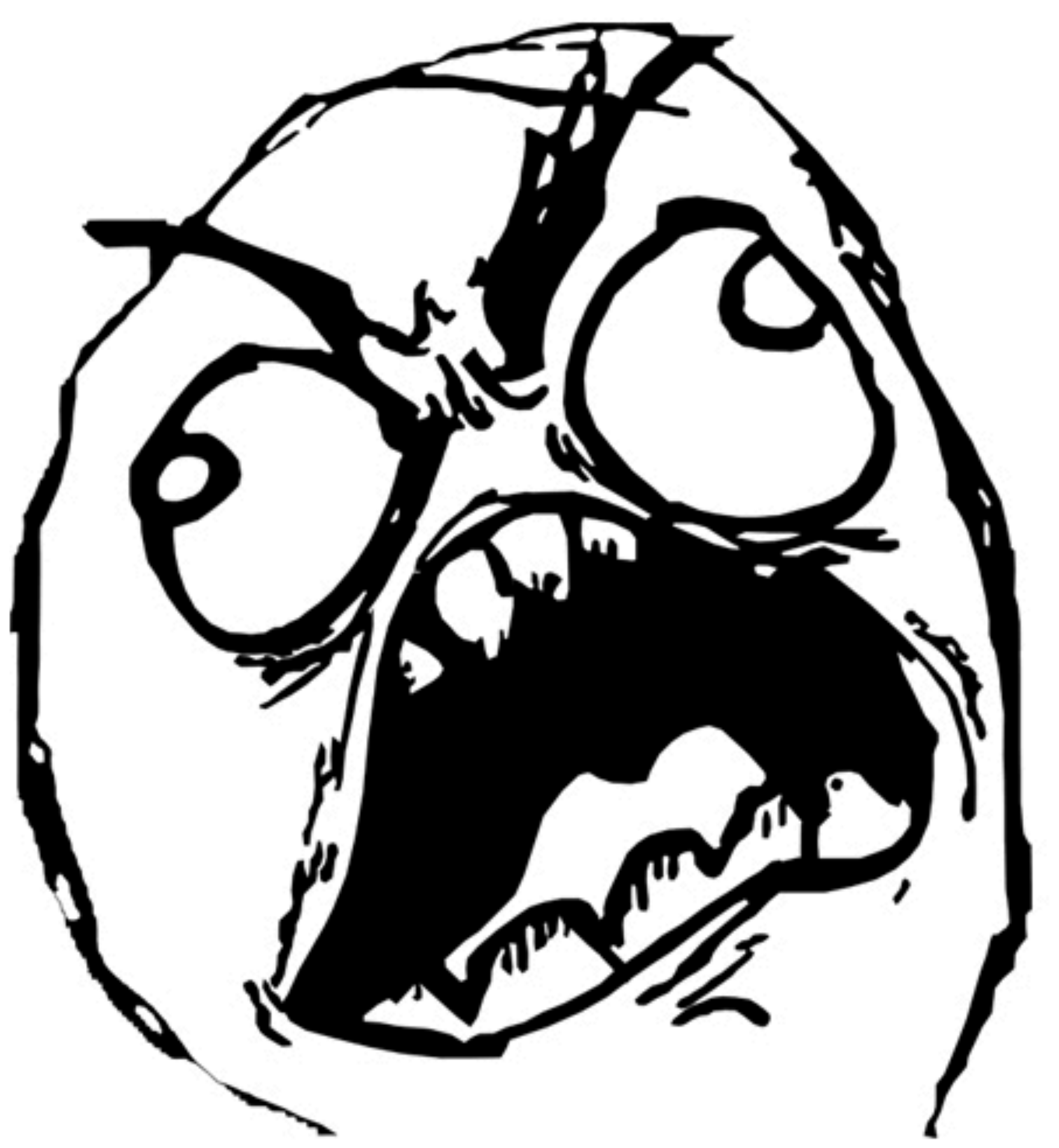


# Database Schemas, Hacker News Edition

> "Do not perform table alter commands. Table alter commands introduce locks and downtimes."

You know what avoids alter? Going schemaless.

```
HACKER_NEWS = ${REPLY_TO_SOMEONE_ON_THE_INTERNET}
```



# Terraform Under The Hood

`terraform plan`

1) Load config file(s): Parse Phase

2) TF Builds a in-mem DAG from config files

3) Refresh State from API calls: Read Calls

PostgreSQL

4) Write refreshed state to statefile

5) Create a DAG from statefile

6) Diffs the DAGs (config and statefile)

7) Save the diffs in the plan

8) User approves plan: `terraform apply`

9) Walk the plan, mutating the state: Delete or Update Calls

10) Refresh State from API calls: Read Calls

11) Write updated state back to state file

# ALTER TABLE: Before Schema

```
$ cat before.tf
resource "postgresql_table" "table2" {
  # provider = "postgresql.pg1"
  name   = "table1"
  schema = "${postgresql_schema.foo_service.name}"

  columns = [
   "${postgresql_column.table2_attribute.id}",
  ]
}

resource "postgresql_column" "table2_attribute" {
  name     = "username"
  type     = "TEXT"
  not_null = false
}
```

# `ALTER TABLE`: After Schema

```
$ cat after.tf
resource "postgresql_table" "table2" {
  # provider = "postgresql.pg1"
  name   = "table1"
  schema = "${postgresql_schema.foo_service.name}"

  columns = [
   "${postgresql_column.table2_attribute.id}",
  ]
}

resource "postgresql_column" "table2_attribute" {
  name     = "username"
  type     = "TEXT"
  not_null = true
  default  = "some default value"
}
```

## That's what it's all about: automatic schema upgrades

Joyent

- This thrust of work started because Terraform has the internal mechanisms to resolve the downtime problems associated with:
    - DDL changes while autovacuum is running
    - Programmatically minimizing the impact to uptime with partial states

# That's what it's all about: automatic schema upgrades

Joyent

This thrust of work started because Terraform has the internal mechanisms to resolve the downtime problems associated with:

- DDL changes while autovacuum is running
- Programmatically minimizing the impact to uptime with partial states

Strategies for each type of attribute update.  Even laborious multi-step, multi-day procedures are good candidates for automation.

PSA: computers are good at this type of work once taught.

# ALTER TABLE: After Schema

```
$ cat before.tf
resource "postgresql_table" "table2" {
  # provider = "postgresql.pg1"
  name   = "table1"
  schema = "${postgresql_schema.foo_service.name}"

  columns = [
   "${postgresql_column.table2_attribute.id}",
  ]
}

resource "postgresql_column" "table2_attribute" {
  name     = "username"
  type     = "TEXT"
  not_null = false
}
```

```
$ cat after.tf
resource "postgresql_table" "table2" {
  # provider = "postgresql.pg1"
  name   = "table1"
  schema = "${postgresql_schema.foo_service.name}"

  columns = [
   "${postgresql_column.table2_attribute.id}",
  ]
}

resource "postgresql_column" "table2_attribute" {
  name     = "username"
  type     = "TEXT"
  not_null = true
  default  = "some default value"
}
```

Impact to DB uptime with naive **ALTER TABLE**

# Terrorform

# Terrorform Tips 1/10

## Design for Blast Radius

"Whoops"
"Sh&^!!!"
"Da*# it!!!"

[[ and much worse ]]

You

# Automation is *automated*

1. Design for Blast Radius

"Stop firing a--holes!" -Darth Helmet
"Stop digging!"

**You to computers**

# Terrorform Tips 3/10

## Think. Plan. Decide. Act. Verify.

1. Design for Blast Radius
2. Automation is automated

"What part of measure twice, cut once didn't you understand?"

"Yup, we have an `ID10T` situation on our hands."

"Explain to me again why you thought that was a good idea?"

"Ca-ching!" -Consultants

**Hopefully not you to coworkers**

IMPORTANT: **`terraform plan`** - **`apply`** cycle is only deterministic with:

```
terraform plan -out=foo.plan
terraform apply foo.plan
```

# Terrorform Tips 3/10

## Think. Plan. Decide. Act. Verify.

1. Design for Blast Radius
2. Automation is automated

*Random sort on above list is ill-advised*

"What part of measure twice, cut once didn't you understand?"

"Yup, we have an `ID10T` situation on our hands."

"Explain to me again why you thought that was a good idea?"

"Ca-ching!" -Consultants

**Hopefully not you to coworkers**

# Terrorform Tips 4/10

## Version Control All The Things

1. Design for Blast Radius
2. Automation is automated
3. Think. Plan. Decide. Act. Verify.

> Space is cheap, context-free understanding is not.
>
> **Me**

# Coordinate Team Activity

1. Design for Blast Radius
2. Automation is automated
3. Think. Plan. Decide. Act. Verify.
4. Version Control All The Things

One driver at a time.

If you need operational concurrency within a service or environment, Terraform may be a bad tool (hint: use a cluster scheduler instead)

Me

# Terrorform Tips 6/10

## Understand "Falling Forward"

A "ROLLBACK" discards work. ROLLBACK for infrastructure is an anti-pattern in thinking/planning. Time moved forward, so did your state. Your error budget was hit with an outage, there is no "ROLLBACK".

Lexicon change: you roll forward to a previously used, known-to-be-good state.

Plan for failure
Plan your escape route
Hope for the happy-path
Know how to put the fire out (or escape from the burning building)

Me

1. Design for Blast Radius
2. Automation is automated
3. Think. Plan. Decide. Act. Verify.
4. Version Control All The Things
5. Coordinate Team Activity

# Terrorform Tips 7/10

## Embrace Layering

1. Design for Blast Radius
2. Automation is automated
3. Think. Plan. Decide. Act. Verify.
4. Version Control All The Things
5. Coordinate Team Activity
6. Understand "Falling Forward"

Small, discrete systems composed in terms of one another are easier to debug.

Simplicity is a virtue.

Because you can doesn't mean you should.

# Terrorform Tips 8/10

## Favor clarity, maintainability, and explicitness

(even at the expense of succinctness)

Two variables with the same value used in two different contexts may need to be separate variables.

This isn't programming "golf."

"Winning" at infrastructure means TCO, flexibility, and uptime.

1. Design for Blast Radius
2. Automation is automated
3. Think. Plan. Decide. Act. Verify.
4. Version Control All The Things
5. Coordinate Team Activity
6. Understand "Falling Forward"
7. Embrace Layering

# Terrorform Tips 9/10

## Refactor in Small Chunks

The first step in fixing a problem is acknowledging you have a problem.

When (not if) you create a fuster-cluck shit-sandwich of a config and go too far in one direction, take time to refine the structure of the code. Favor maintenance and explicitness.

1. Design for Blast Radius
2. Automation is automated
3. Think. Plan. Decide. Act. Verify.
4. Version Control All The Things
5. Coordinate Team Activity
6. Understand "Falling Forward"
7. Embrace Layering
8. Favor clarity, maintainability, and explicitness

## Terrorform Tips 10/10

# Make Mistakes, Get Messy

New game: like the fortune cookie game where you read the fortune cookie and append, "in bed," do the same thing but use "in dev."

There is no compression algorithm for experience.
-Werner Vogels

1. Design for Blast Radius
2. Automation is automated
3. Think. Plan. Decide. Act. Verify.
4. Version Control All The Things
5. Coordinate Team Activity
6. Understand "Falling Forward"
7. Embrace Layering
8. Favor clarity, maintainability, and explicitness
9. Refactor in Small Chunks

# Terrorform Tips

## This advice is worth as much as you paid for it.

Lawyer alert: Neither me or employer warranty this advice nor are we responsible for your recklessness, bugs you will encounter, or create on your own.

1. Design for Blast Radius
2. Automation is automated
3. Think. Plan. Decide. Act. Verify.
4. Version Control All The Things
5. Coordinate Team Activity
6. Understand "Falling Forward"
7. Embrace Layering
8. Favor clarity, maintainability, and explicitness
9. Refactor in Small Chunks
10. Make Mistakes, Get Messy (in dev)

# State of the World: May 2017

| Done | In Process | Backlog |
|---|---|---|
| `postgresql_database` | `postgresql_sequence` | `postgresql_select` (data source) |
| `postgresql_extension` | `postgresql_column` | `postgresql_view` |
| `postgresql_role` | `postgresql_table` | `postgresql_pl` |
| `postgresql_schema` | ^^ may be done by the end of this conference (unit of measurement for completion is being measured in hours) | `postgresql_database` (settings) |
| | | `postgresql_role` (settings) |
| | | `postgresql_index` |
| | | `postgresql_table_space` |
| | | `postgresql_dml` |
| | | `postgresql_type` |
| | | `postgresql_fdw` |
| | | `postgresql_table` (*partition syntax) |

## Parting Thoughts

- Don't drive automation unless you understand Terraform is evaluating a dependency graph
- `terraform plan`
- Pause and read the output
- Understand the change
- Keep tweaking and re-planning until you understand the change
- `terraform apply` once everything looks right
- With great power comes great responsibility
- Don't let Terraform become Terrorform
- Only you can prevent forrest fires

# Thank you! Questions?

```
https://github.com/sean-/pgcon-2017
https://www.slideshare.net/SeanChittenden/codified-postgresql-schema
```

**Sean Chittenden**
*Engineering*

seanc@joyent.com
@SeanChittenden