Writing a Go application with PostgreSQL using pgx

Henrique Vicente

Go meetup @ HATCH Amsterdam, 12 April 2022

Trajectory to PostgreSQL

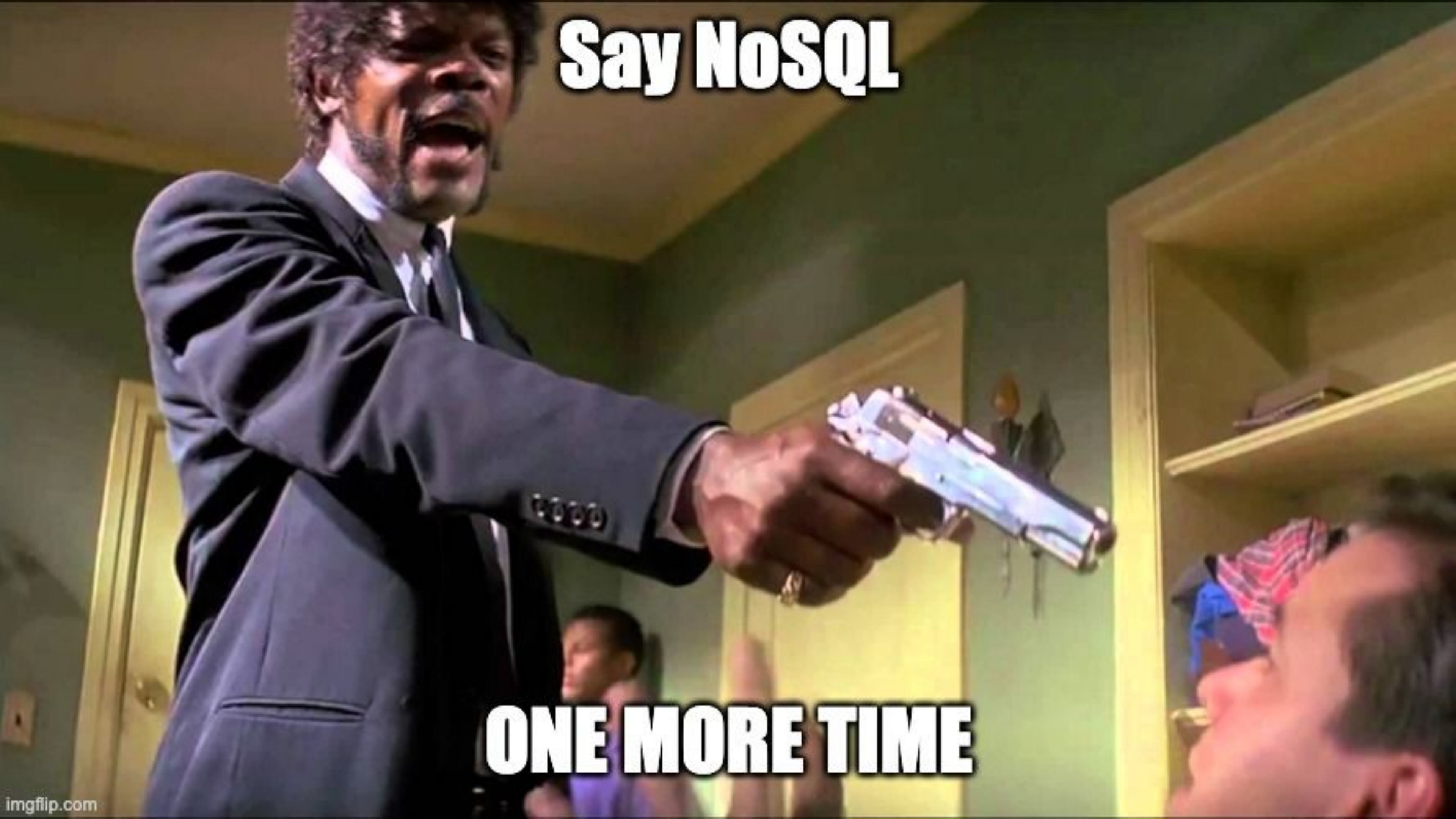
Migration from a document-oriented database to
PostgreSQL (RDS) + OpenSearch (Amazon's fork of Elasticsearch)

Problems with previous database

- Poor developer experience
- Reliability and performance issues
- High cost: support, licensing, and operational
- Vendor lock-in
- Steep learning curve

Why PostgreSQL

- Object-Relational Database Management System
- Almost 100% SQL standards-compliant
- Excellent for client/server architecture
- **JSON types**
- Custom data types
- Inheritance between tables


Say NoSQL
ONE MORE TIME
imgflip.com

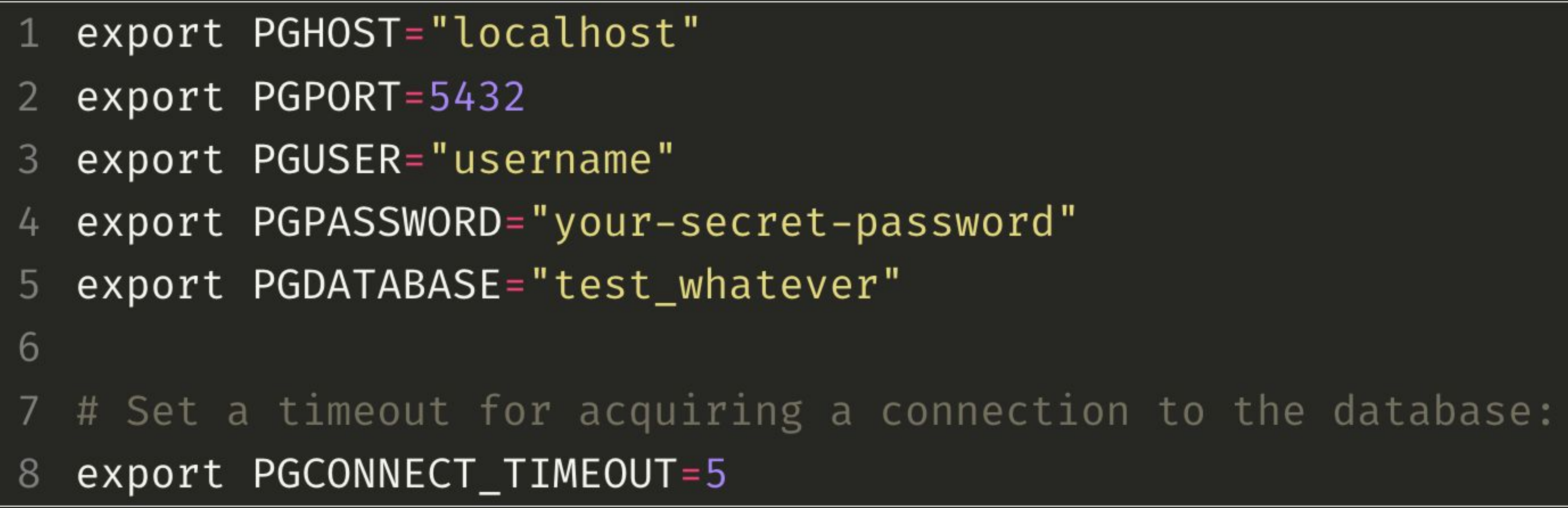
PostgreSQL advantages

- Great developer experience
- Reliability
- Performance
- Open-source software without vendor lock-in
- Community and multiple support providers
- Works everywhere: even on a new M1 MacBook

PostgreSQL drawbacks

- Plan schema up-front (wait, isn't this an advantage?)
- Full-text search capabilities is limited (but we've got it covered).

Environment variables for configuring PostgreSQL https://www.postgresql.org/docs/current/libpq-envars.html



```
export PGHOST="localhost"
export PGPORT=5432
export PGUSER="username"
export PGPASSWORD="your-secret-password"
export PGDATABASE="test_whatever"

# Set a timeout for acquiring a connection to the database:
export PGCONNECT_TIMEOUT=5
```

Environment variables might be problematic due to security concerns, so be careful.

- Globals
- Easy to leak to child processes or metrics probes

Also:

- Process --flags are as bad!
- Can you point to secrets, rather than expose them?

Tip: direnv makes it very convenient to use it on a development environment.

database/sql vs. pgx interface

Talking to SQL databases in Go, you almost always want to use a database through database/sql.

It provides a nice common interface, and makes supporting different databases easier.

PostgreSQL drivers for database/sql:

- github.com/jackc/pgx
- github.com/lib/pq (discontinued in favor of pgx)

database/sql vs. pgx interface

Among better performance, by using pgx interface you've:

- Support for approximately 70 different PostgreSQL types
- Automatic statement preparation and caching
- Batch queries
- Conversion of PostgreSQL arrays to Go slice mapping
- JSON and JSONB support
- COPY protocol support

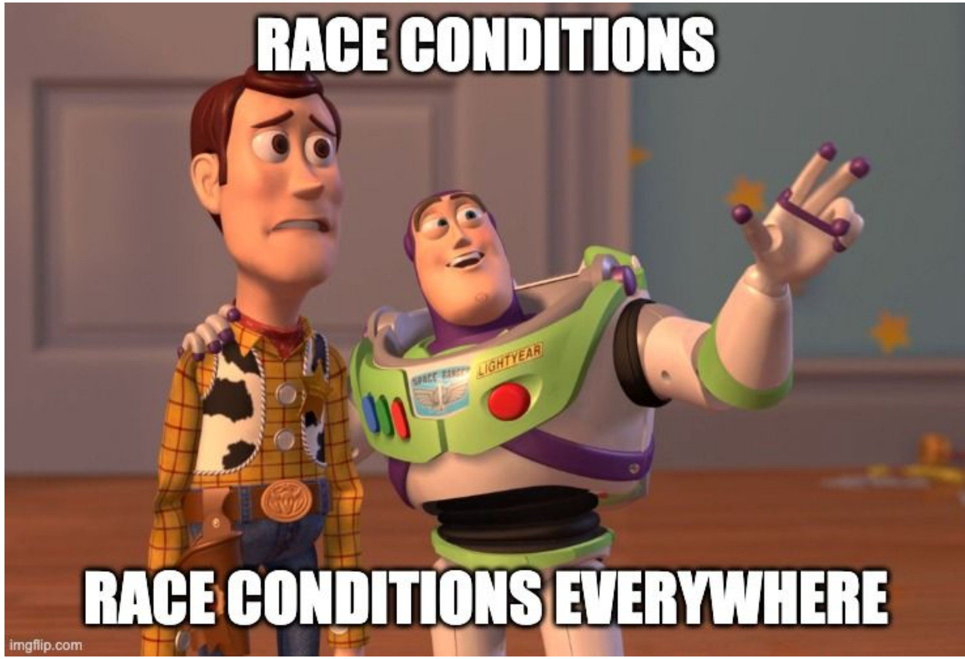
database/sql is limited to int64, float64, bool, []byte, string, time.Time, or nil

Concurrency: pgxpool

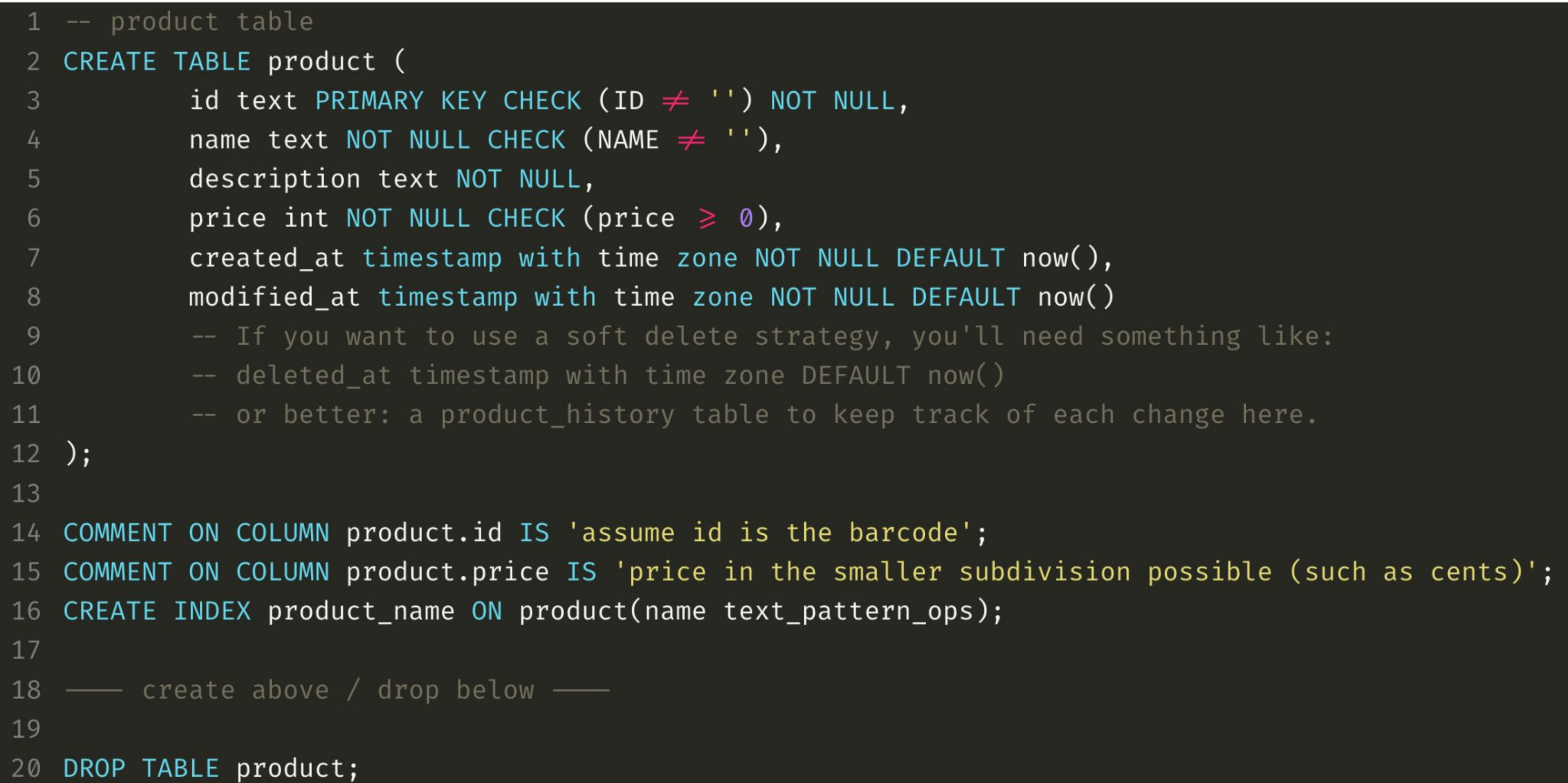
- pgx.Conn is a low-level implementation
- Use pgxpool.Conn to access the database concurrently

Default:

conf.MaxConns = runtime.NumCPU()



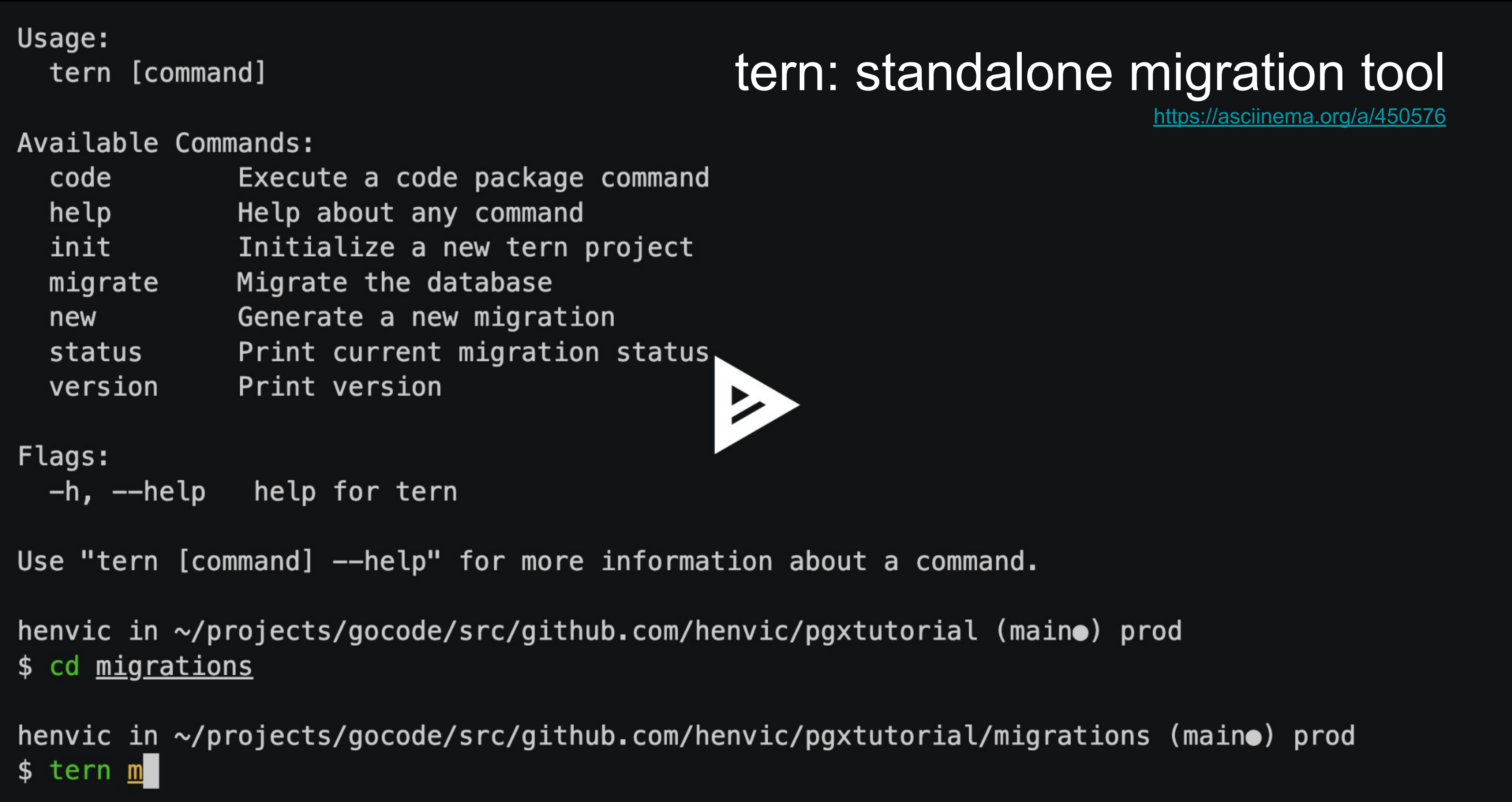
Database migrations: 001_initial_schema.sql, 002_…



```sql
-- product table
CREATE TABLE product (
        id text PRIMARY KEY CHECK (ID ≠ '') NOT NULL,
        name text NOT NULL CHECK (NAME ≠ ''),
        description text NOT NULL,
        price int NOT NULL CHECK (price ⩾ 0),
        created_at timestamp with time zone NOT NULL DEFAULT now(),
        modified_at timestamp with time zone NOT NULL DEFAULT now()
        -- If you want to use a soft delete strategy, you'll need something like:
        -- deleted_at timestamp with time zone DEFAULT now()
        -- or better: a product_history table to keep track of each change here.
);

COMMENT ON COLUMN product.id IS 'assume id is the barcode';
COMMENT ON COLUMN product.price IS 'price in the smaller subdivision possible (such as cents)';
CREATE INDEX product_name ON product(name text_pattern_ops);

—— create above / drop below ——

DROP TABLE product;
```

tern: standalone migration tool

https://asciinema.org/a/450576

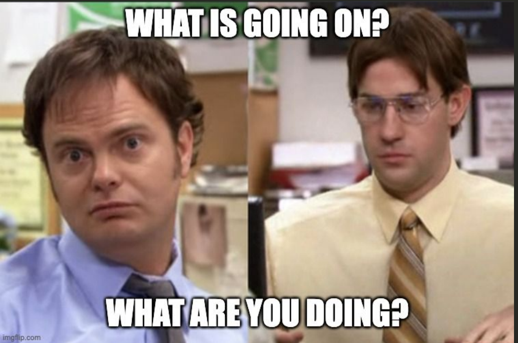


```
Usage:
  tern [command]

Available Commands:
  code         Execute a code package command
  help         Help about any command
  init         Initialize a new tern project
  migrate      Migrate the database
  new          Generate a new migration
  status       Print current migration status
  version      Print version

Flags:
  -h, --help   help for tern

Use "tern [command] --help" for more information about a command.

henvic in ~/projects/gocode/src/github.com/henvic/pgxtutorial (main●) prod
$ cd migrations

henvic in ~/projects/gocode/src/github.com/henvic/pgxtutorial/migrations (main●) prod
$ tern m
```

Database layer

It might be useful to define an interface, even if bothersome.

What about mocks?



```go
// DB layer.
//go:generate mockgen --build_flags=--mod=mod -package inventory -destination mock_db_test.go . DB
type DB interface {
	// CreateProduct creates a new product.
	CreateProduct(ctx context.Context, params CreateProductParams) error

	// UpdateProduct updates an existing product.
	UpdateProduct(ctx context.Context, params UpdateProductParams) error

	// GetProduct returns a product.
	GetProduct(ctx context.Context, id string) (*Product, error)

	// ...

	// TransactionContext returns a copy of the parent context which begins a transaction
	// to PostgreSQL.
	TransactionContext(ctx context.Context) (context.Context, error)

	// Commit transaction from context.
	Commit(ctx context.Context) error

	// Rollback transaction from context.
	Rollback(ctx context.Context) error

	// WithAcquire returns a copy of the parent context which acquires a connection
	// to PostgreSQL from pgxpool to make sure commands executed in series reuse the
	// same database connection.
	WithAcquire(ctx context.Context) (dbCtx context.Context, err error)

	// Release PostgreSQL connection acquired by context back to the pool.
	Release(ctx context.Context)
}
```

Testing strategy

Get superior returns from your tests:

- Favor writing integration test with a real implementation
- Use test doubles to test scenarios of database failure.

Real implementation	**Test doubles (mocks, fakes…)**
SELECT returning data or not found.	Database connection issue.
INSERT, UPDATE, etc.	Simulation of unexpected DB error.
Constraints checks.	Expensive operation (ask why first)

Inspiration: Software Engineering at Google: Lessons Learned from Programming Over Time
https://amzn.to/3uqZWP7

Just like Theranos' blood tests…

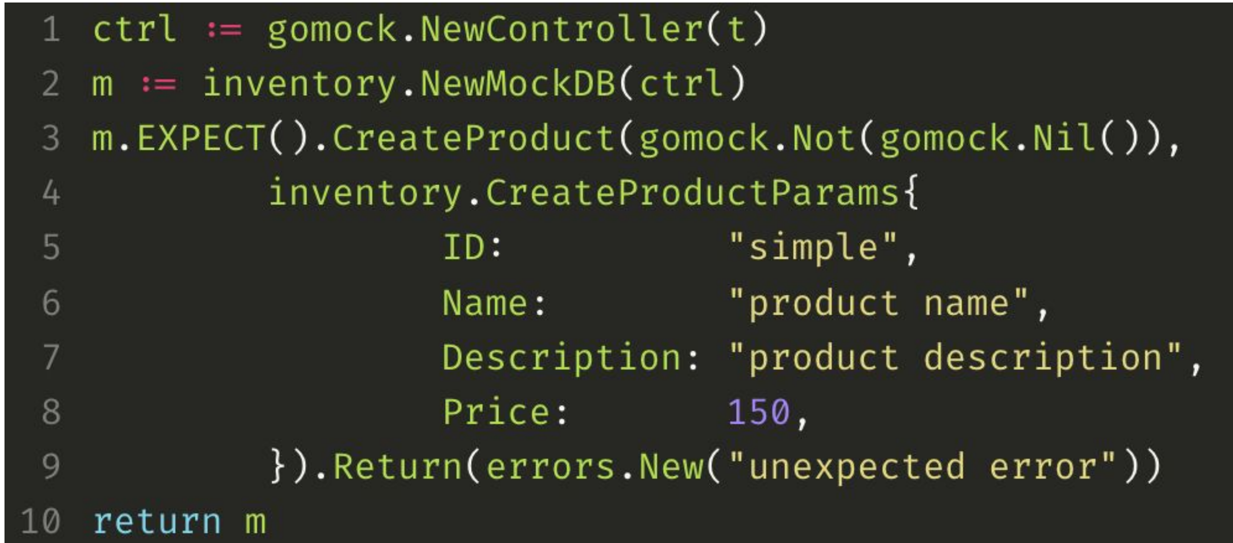
Testing interfaces for the sake of it is overrated.

Some tools

- GoMock github.com/golang/mock
- go-cmp github.com/google/go-cmp
- scany github.com/georgysavva/scany
- pgtools github.com/hatch-studio/pgtools
- squirrel github.com/Masterminds/squirrel (I prefer plain SQL, though)

GoMock

github.com/golang/mock

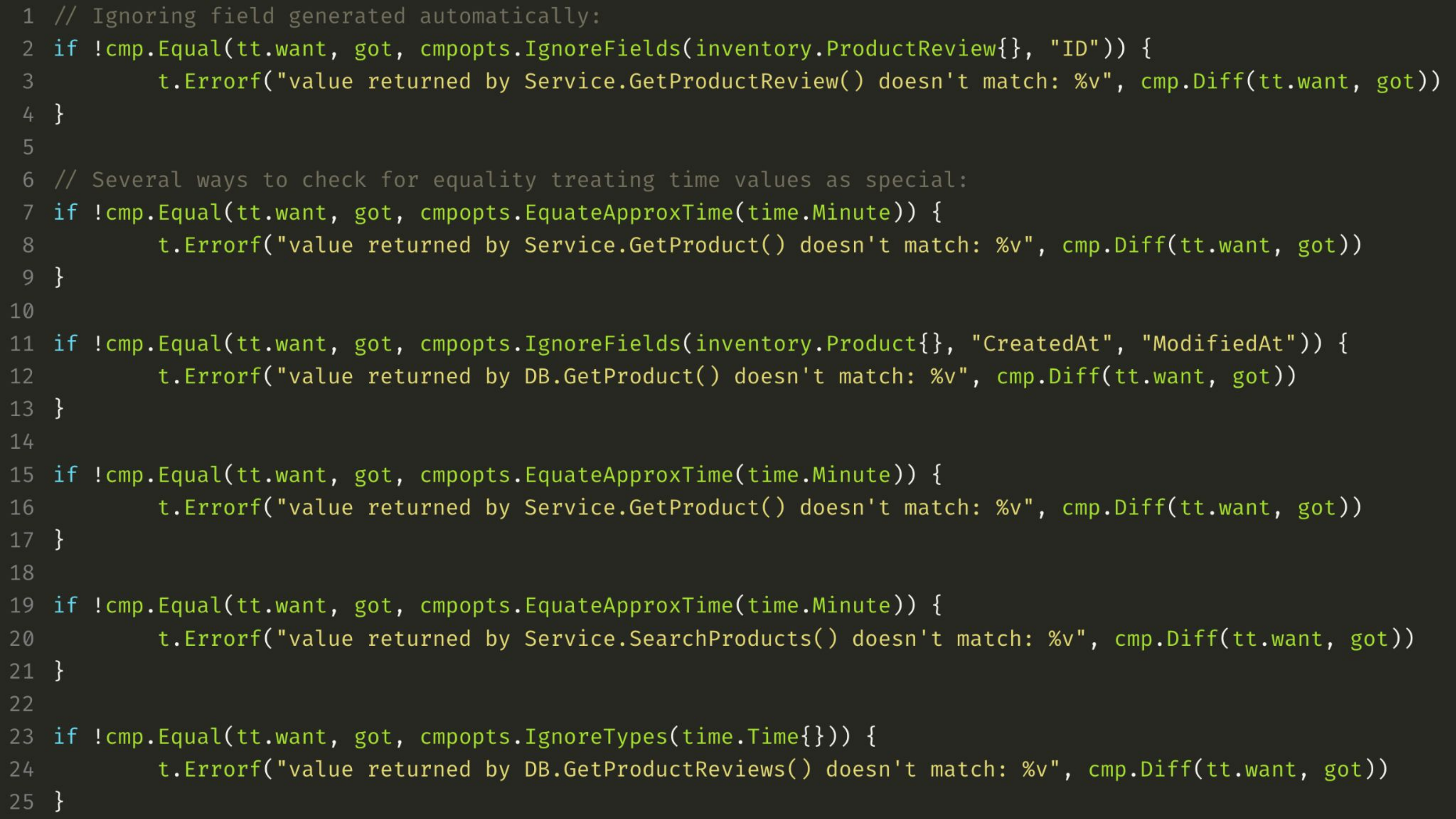


```go
ctrl := gomock.NewController(t)
m := inventory.NewMockDB(ctrl)
m.EXPECT().CreateProduct(gomock.Not(gomock.Nil()),
        inventory.CreateProductParams{
                ID:          "simple",
                Name:        "product name",
                Description: "product description",
                Price:       150,
        }).Return(errors.New("unexpected error"))
return m
```

Comparing values with go-cmp

github.com/google/go-cmp

- *A more powerful and safer alternative to reflect.DeepEqual for comparing whether two values are semantically equal.*



```go
// Ignoring field generated automatically:
if !cmp.Equal(tt.want, got, cmpopts.IgnoreFields(inventory.ProductReview{}, "ID")) {
	t.Errorf("value returned by Service.GetProductReview() doesn't match: %v", cmp.Diff(tt.want, got))
}

// Several ways to check for equality treating time values as special:
if !cmp.Equal(tt.want, got, cmpopts.EquateApproxTime(time.Minute)) {
	t.Errorf("value returned by Service.GetProduct() doesn't match: %v", cmp.Diff(tt.want, got))
}

if !cmp.Equal(tt.want, got, cmpopts.IgnoreFields(inventory.Product{}, "CreatedAt", "ModifiedAt")) {
	t.Errorf("value returned by DB.GetProduct() doesn't match: %v", cmp.Diff(tt.want, got))
}

if !cmp.Equal(tt.want, got, cmpopts.EquateApproxTime(time.Minute)) {
	t.Errorf("value returned by Service.GetProduct() doesn't match: %v", cmp.Diff(tt.want, got))
}

if !cmp.Equal(tt.want, got, cmpopts.EquateApproxTime(time.Minute)) {
	t.Errorf("value returned by Service.SearchProducts() doesn't match: %v", cmp.Diff(tt.want, got))
}

if !cmp.Equal(tt.want, got, cmpopts.IgnoreTypes(time.Time{})) {
	t.Errorf("value returned by DB.GetProductReviews() doesn't match: %v", cmp.Diff(tt.want, got))
}
```

scany

```go
package main

import (
	"context"

	"github.com/jackc/pgx/v4/pgxpool"

	"github.com/georgysavva/scany/pgxscan"
)

type User struct {
	ID    string
	Name  string
	Email string
	Age   int
}

func main() {
	ctx := context.Background()
	db, _ := pgxpool.Connect(ctx, "example-connection-url")

	var users []*User
	pgxscan.Select(ctx, db, &users, `SELECT id, name, email, age FROM users`)
	// users variable now contains data from all rows.
}
```

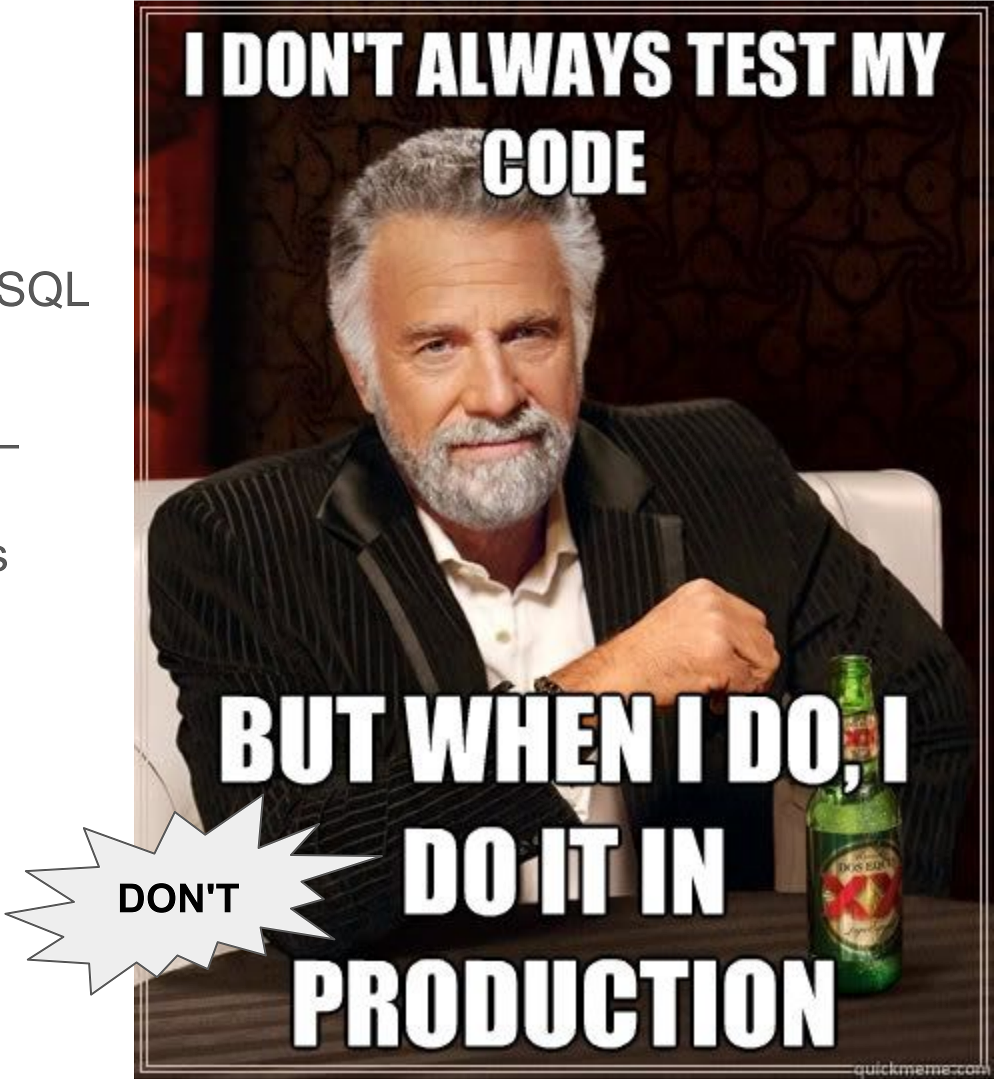
pgtools

github.com/hatch-studio/pgtools contains:

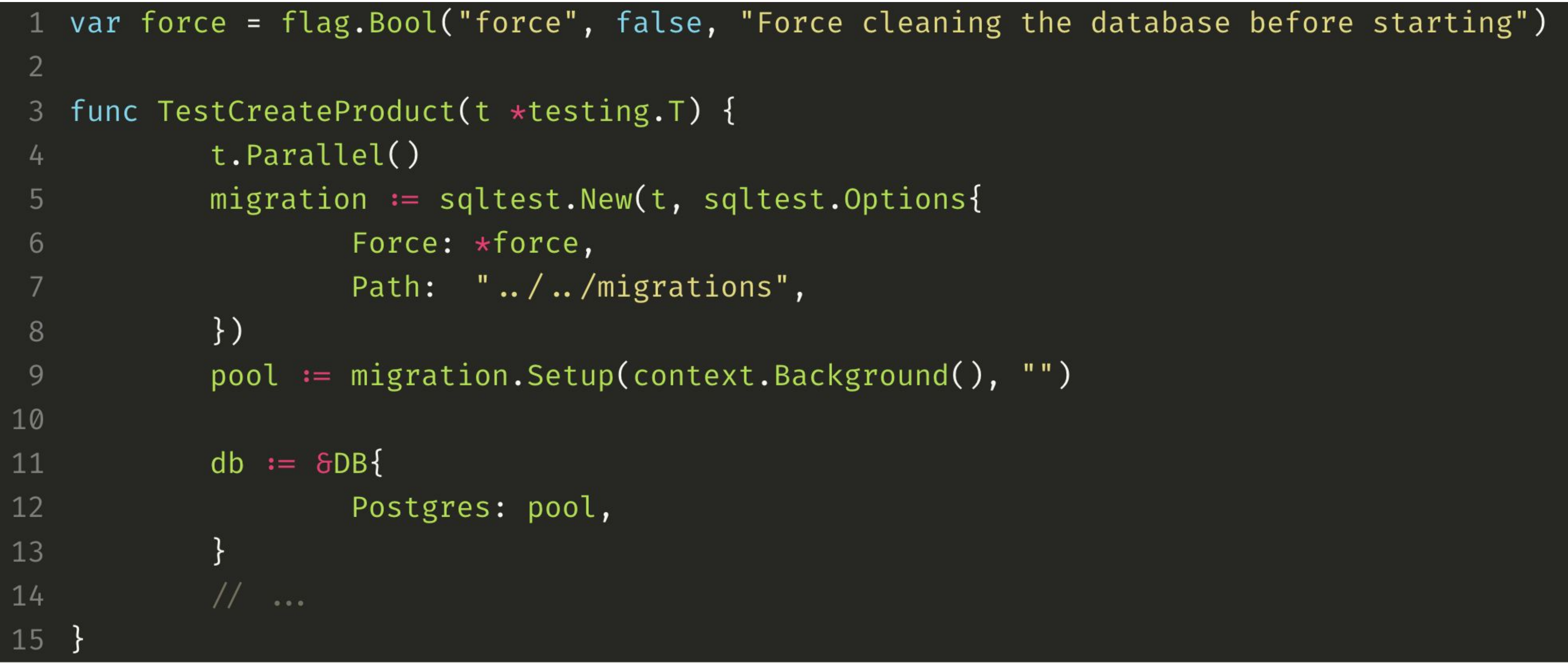
- Package to make creating SELECT for writing queries easier.
- Package sqltest, which makes writing integration tests very efficient.

Tests with sqltest

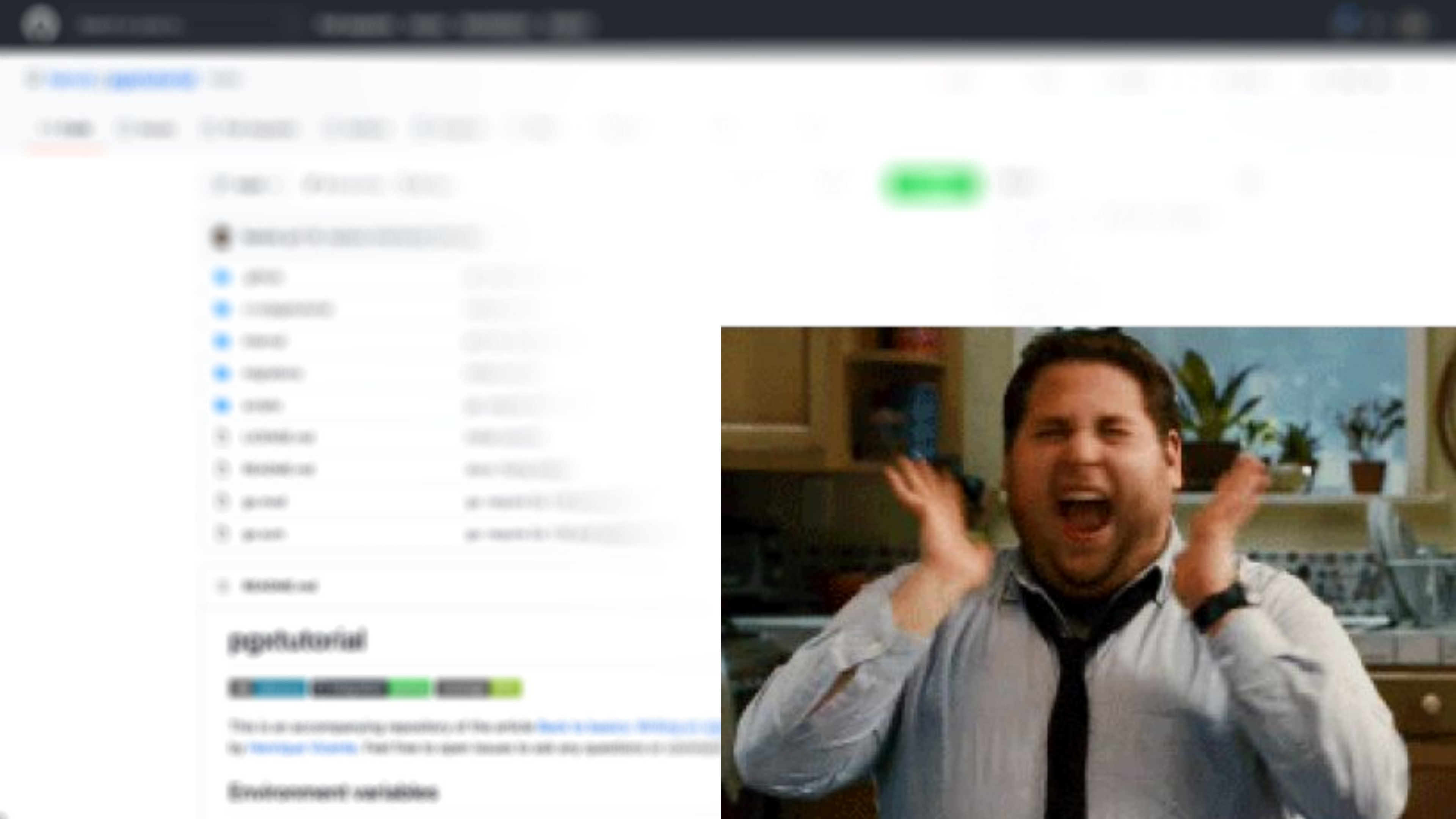
- Auto-discovery recognizes PostgreSQL configuration automatically.
- Creates a database with prefix test_ (safety) + normalized test names.
- Your tests can run in parallel just as fine.


I DON'T ALWAYS TEST MY CODE
BUT WHEN I DO, I DO IT IN PRODUCTION
DON'T
quickmeme.com

sqltest + tern = easy tests https://asciinema.org/a/450576

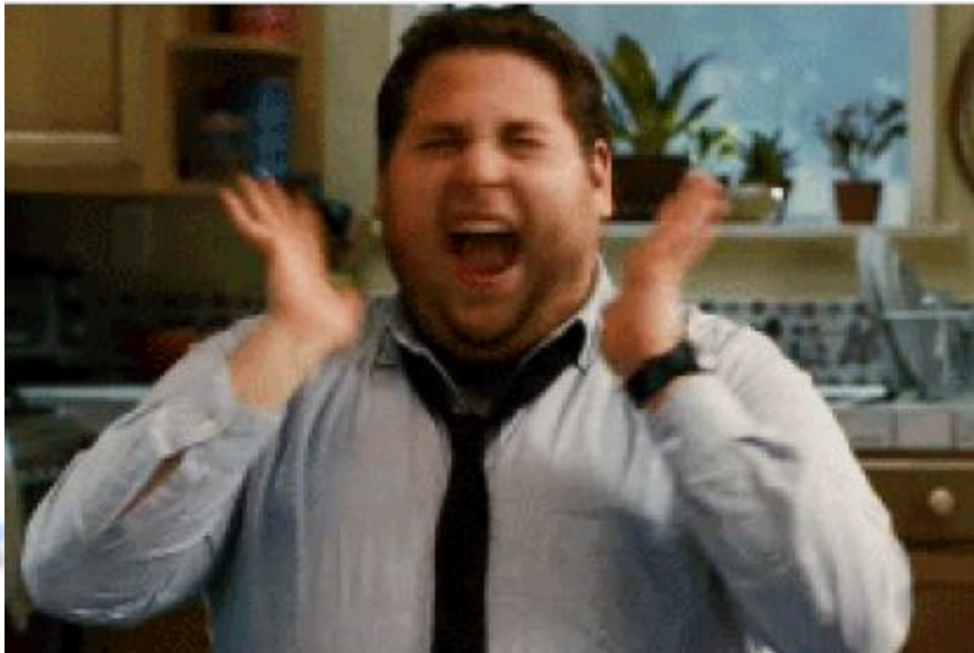


```go
var force = flag.Bool("force", false, "Force cleaning the database before starting")

func TestCreateProduct(t *testing.T) {
	t.Parallel()
	migration := sqltest.New(t, sqltest.Options{
		Force: *force,
		Path:  "../../migrations",
	})
	pool := migration.Setup(context.Background(), "")

	db := &DB{
		Postgres: pool,
	}
	// ...
}
```

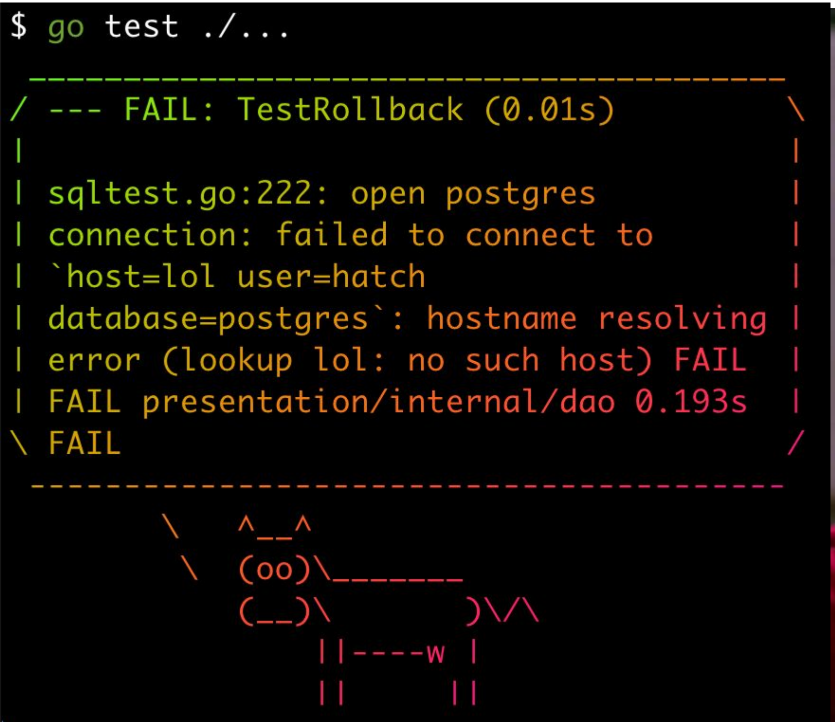


pgetutorial

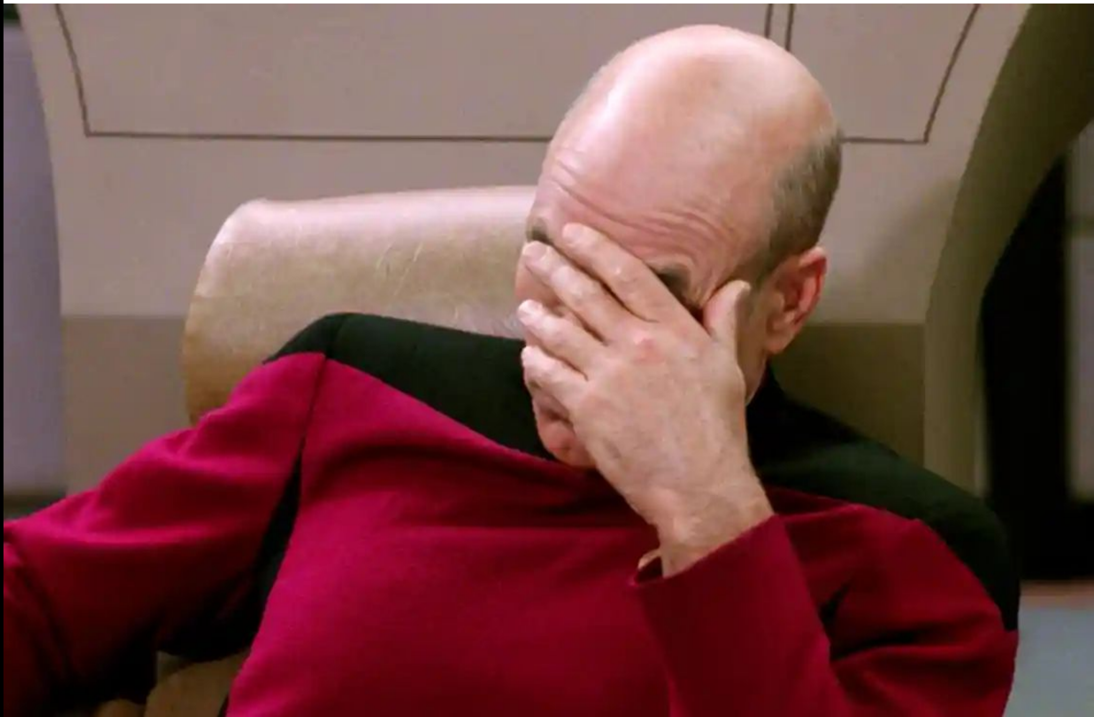
Environment variables



Then you discover running its integration tests is hard.



```
$ go test ./...
 ----------------------------------------
/ --- FAIL: TestRollback (0.01s)          \
|                                         |
| sqltest.go:222: open postgres           |
| connection: failed to connect to        |
| `host=lol user=hatch                    |
| database=postgres`: hostname resolving |
| error (lookup lol: no such host) FAIL  |
| FAIL presentation/internal/dao 0.193s  |
\ FAIL                                    /
 ----------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```



$ go test ./…

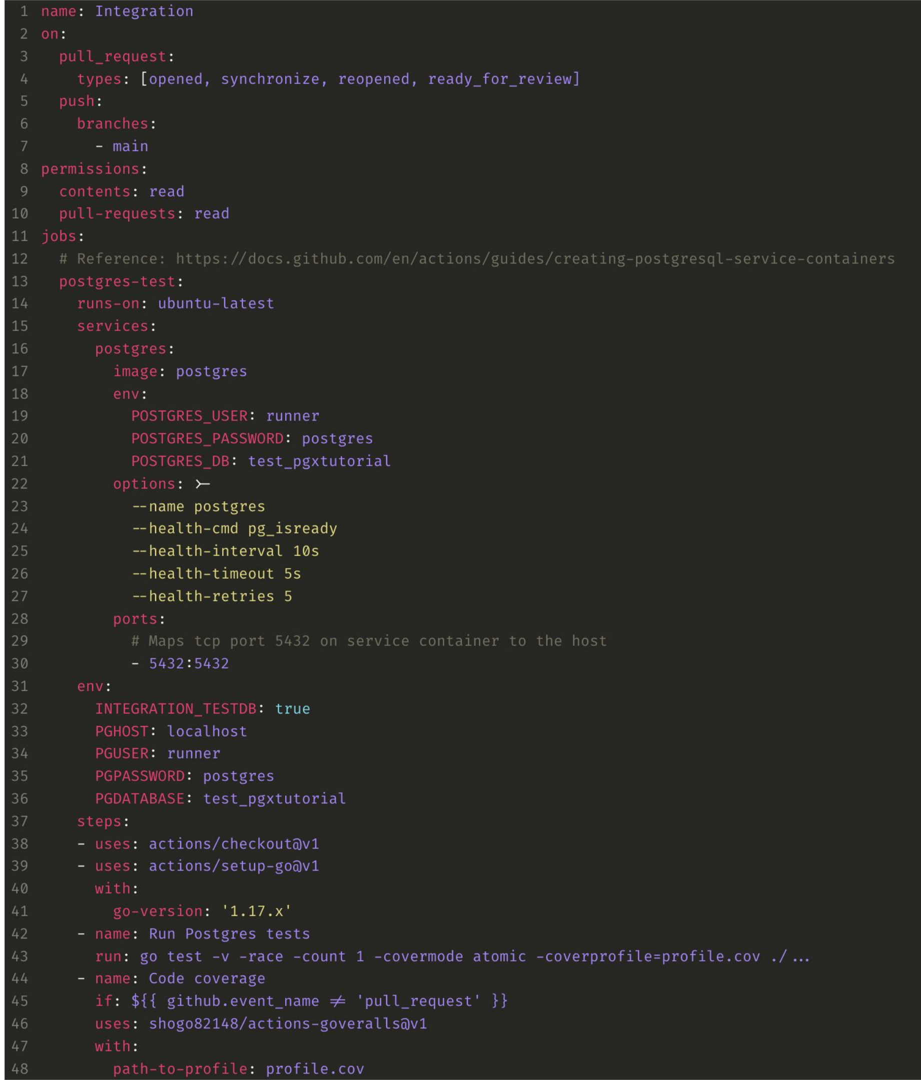
From zero to running running integration tests in few seconds. Try it:

1. Install or configure PostgreSQL environment variables so that psql works.
2.
```
$ export INTEGRATION_TESTDB=true
$ go test -v ./...
```

Continuous Integration

Run your DB on any CI/CD system.

For example, with a pull request workflow on GitHub Actions, stateless.



```yaml
name: Integration
on:
  pull_request:
    types: [opened, synchronize, reopened, ready_for_review]
  push:
    branches:
      - main
permissions:
  contents: read
  pull-requests: read
jobs:
  # Reference: https://docs.github.com/en/actions/guides/creating-postgresql-service-containers
  postgres-test:
    runs-on: ubuntu-latest
    services:
      postgres:
        image: postgres
        env:
          POSTGRES_USER: runner
          POSTGRES_PASSWORD: postgres
          POSTGRES_DB: test_pgxtutorial
        options: >-
          --name postgres
          --health-cmd pg_isready
          --health-interval 10s
          --health-timeout 5s
          --health-retries 5
        ports:
          # Maps tcp port 5432 on service container to the host
          - 5432:5432
    env:
      INTEGRATION_TESTDB: true
      PGHOST: localhost
      PGUSER: runner
      PGPASSWORD: postgres
      PGDATABASE: test_pgxtutorial
    steps:
    - uses: actions/checkout@v1
    - uses: actions/setup-go@v1
      with:
        go-version: '1.17.x'
    - name: Run Postgres tests
      run: go test -v -race -count 1 -covermode atomic -coverprofile=profile.cov ./...
    - name: Code coverage
      if: ${{ github.event_name != 'pull_request' }}
      uses: shogo82148/actions-goveralls@v1
      with:
        path-to-profile: profile.cov
```

```
$ git clone https://github.com/henvic/pgxtutorial.git
$ cd pgxtutorial
```

```
# Run tests with:
$ INTEGRATION_TESTDB=true go test -v ./...
```

```
# To execute application, first create a database and run migrations.
$ psql -c "CREATE DATABASE pgxtutorial;"
$ export PGDATABASE=pgxtutorial
$ tern migrate -m ./migrations
# Then, run it (without installing):
$ go run ./cmd/pgxtutorial
2021/11/22 07:21:21 HTTP server listening at localhost:8080
2021/11/22 07:21:21 gRPC server listening at 127.0.0.1:8082
```

Proof it works https://asciinema.org/a/450580

```
api.Inventory@localhost:8082> call SearchProducts
query_string (TYPE_STRING) => chair
✔ min_price
min_price (TYPE_INT64) => 1
✔ max_price
max_price (TYPE_INT64) => 1000
✔ page
page (TYPE_INT32) => 1
{
  "items": [
    {
      "description": "This is a nice chair.",
      "id": "chair",
      "name": "Nice office chair",
      "price": "14"
    }
  ],
  "total": 1
}

api.Inventory@localhost:8082> █
```

Takeaways

- PostgreSQL is good
- There are many tools to help you
- Balance between integration and unit tests
- Continuous Integration is easier than you think

Thanks. Questions?

Henrique Vicente, HATCH Studio

- Article: https://henvic.dev/posts/go-postgres/
- Repository: https://github.com/henvic/pgxtutorial