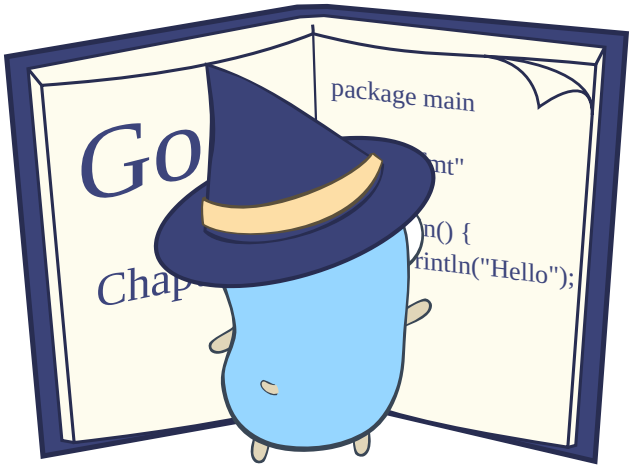
A Journey to Postgres Productivity

13th of November 2020

Johan Brandhorst
Buf

Today we will

- Talk about state management
- Talk about databases
- Look at some tooling
- Dive into some practical examples
- Become Postgres rockstars



Witch by Egon Elbre (https://twitter.com/egonelbre)

State management

- The reason we can't have nice things
- The reason we can have some things

More Lego by Xavi Cabrera (https://unsplash.com/photos/kn-UmDZQDjM)

In memory

- Easy
- Fast access

```
package user

type User struct {
    Name string
    Age  uint
}

users := []User{
    {
        Name: "Johan",
        Age:  29,
    },
}
```

- What if we want to update the application?
- What if we want run several applications?

The filesystem

- Persists through restarts
- Can be shared between processes

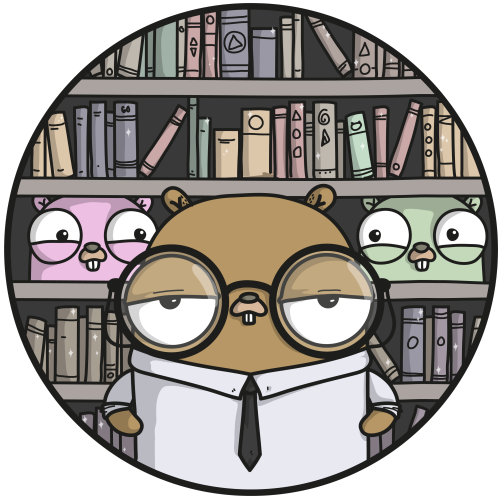
```
marshalled := fmt.Sprintf("%s, %d", users[0].Name, users[0].Age)

err := ioutil.WriteFile("./users.csv", []byte(marshalled), 0644)
if err != nil {
    // handle err
}
```

- Marshaling format?
- Reading data efficiently?
- Filtering results?

Data stores

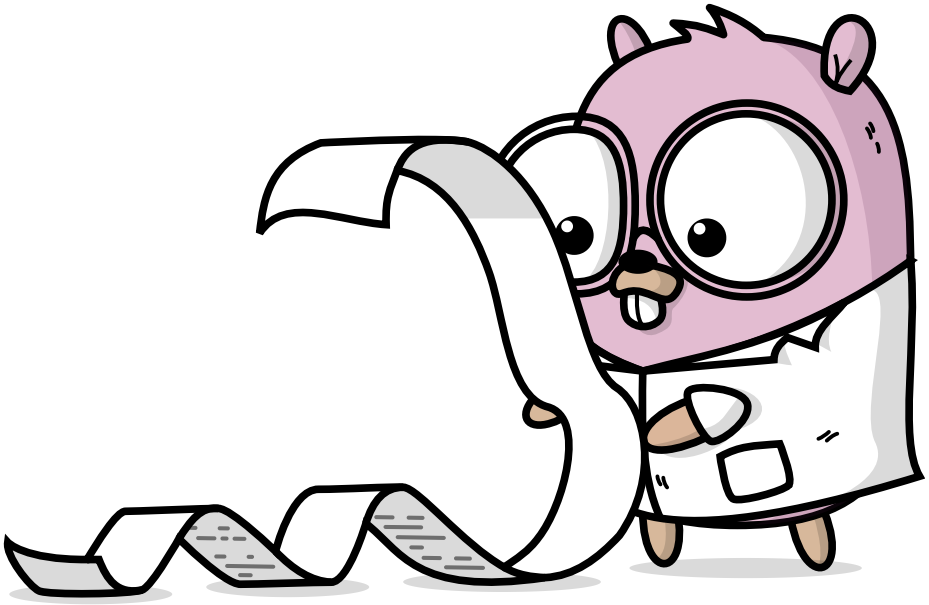
- Persisting data
- Storing large amounts of data
- Concurrent access
- Filtering results
- Much more...



Nerdy by Ashley Willis (https://twitter.com/ashleymcnamara)

Data storage solutions

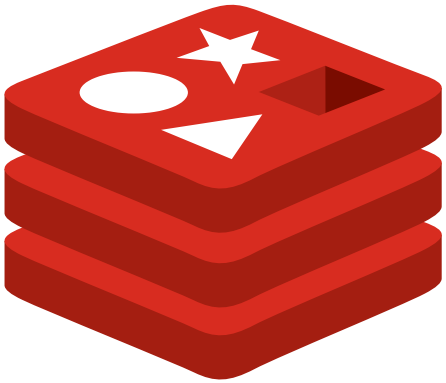
- In-memory key-value stores
- Document stores
- Relational databases



GopherNotes Gopher by Marcus Olsson (https://twitter.com/marcusolsson)

In-memory Key-value stores

- Good for caches
- Very fast
- Less structured
- Limited to size of memory
- Memcached, Redis



redis

Source: Redis (https://redis.io/)

Document stores

- NoSQL
- Filesystem
- Fast
- Unstructured
- Some compromise safety for speed
- MongoDB, Couchbase, ElasticSearch

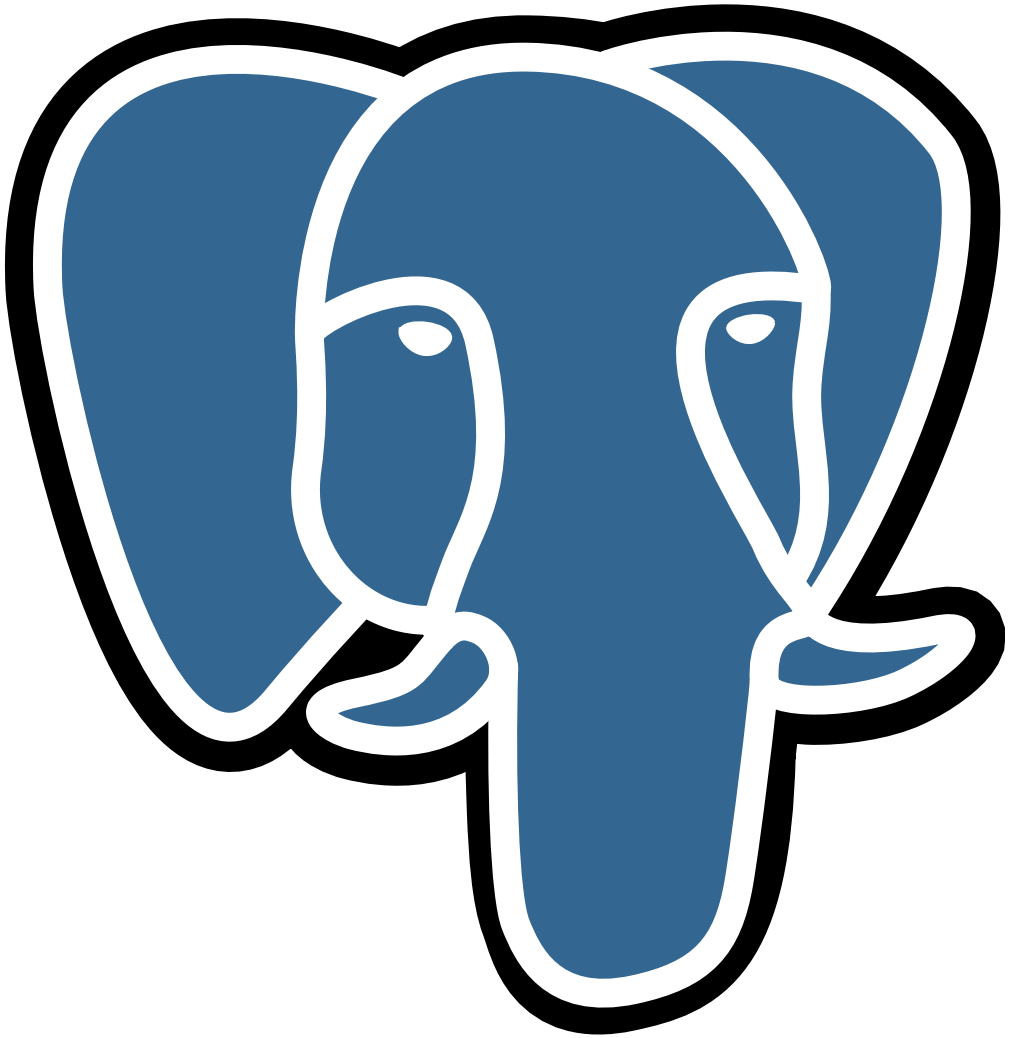
> ***MongoDB lost data and violated causal [sic] by default.*** *Somehow that became "among the strongest data consistency, correctness, and safety guarantees of any database available today"!*
>
> - *@jepsen_io (https://twitter.com/jepsen_io/status/1255867265997844484)*

Relational databases

- SQL for writing and reading
- Filesystem
- Consistency guarantees
- ACID transactions
- Oracle, Microsoft SQL Server, MariaDB, MySQL, PostgreSQL, CockroachDB

Source: MySQL (https://www.mysql.com/)

What should I use?



Source: PostgreSQL (https://www.postgresql.org/)

Postgres

- Proven track record
- Free
- Open Source
- Good Go library support
- Need scale? CockroachDB
- Good cloud support

Ok by Takuya Ueda (https://github.com/tenntenn)

database/sql

- General purpose SQL abstraction
- Requires specialized drivers
- Built-in connection pooling
- Special types for NULL handling

```
import _ "github.com/yourdatabase/driver"

rows, err := db.QueryContext(ctx, "SELECT name FROM users WHERE age = $1", age)
if err != nil {
    log.Fatal(err)
}
defer rows.Close()
for rows.Next() {
    var name string
    if err := rows.Scan(&name); err != nil {
        log.Fatal(err)
    }
}
err = rows.Err()
// handle err
```

Problems with standard library

- Enforced side-effect-by-import
- Not fully type safe
- (*sql.Rows).Close() easily forgotten/misused
- Misuse can lead to SQL injection vulnerabilities

```
import _ "github.com/yourdatabase/driver" // Blank import forces driver to use init()
//                                ,-- careful not to let user control any part of this
rows, err := db.QueryContext(ctx, "SELECT name FROM users WHERE age = $1", age)
if err != nil { //                                                    ^-- interface{} type
    log.Fatal(err)
}
defer rows.Close() // Don't forget to check this error or you might miss a failure
for rows.Next() {
    var name string
    if err := rows.Scan(&name); err != nil {
        log.Fatal(err)
    }
}
err = rows.Err() // Don't forget to call this!
// handle err
```

Postgres tooling

- ORMs
- Generators
- Query builders
- Drivers
- Migrations
- Testing

Question by Takuya Ueda (https://github.com/tenntenn)

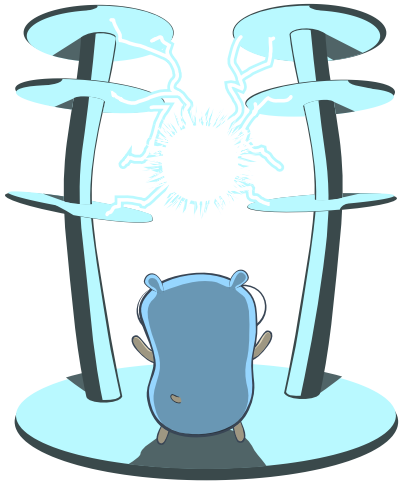
Object-relational Mapping (ORM)

- Hides the database behind user friendly API
- Can lead to inefficient queries
- Not a great fit without generics
- Can't expose all database features
- No need (or opportunity) to learn SQL

Witch Too Much Candy by Egon Elbre (https://twitter.com/egonelbre)

Generator libraries

- Hides the database behind generated types and functions
- Type safe interfaces
- Requires source markup
- Often limited in their database support



Power To The Masses by Egon Elbre (https://twitter.com/egonelbre)

Query builders

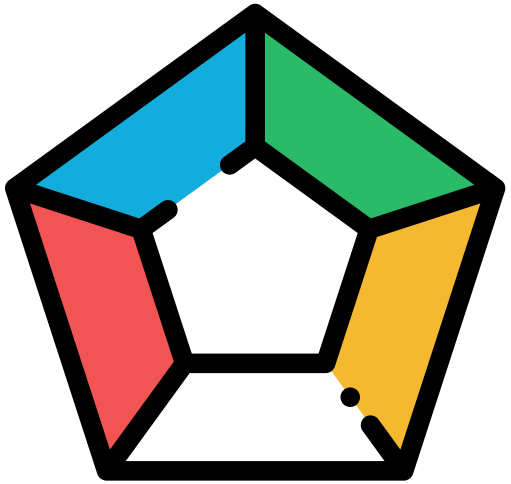
- Fluent, builder-pattern interface
- Easy to use
- Automatically parametrises queries
- Not always type safe

Gopher Rocks by Ashley Willis (https://twitter.com/ashleymcnamara)

What tooling should we use?

The five pillars of our Postgres stack

- Driver (jackc/pgx)
- Migrations (golang-migrate/migrate)
- Query generator (kyleconroy/sqlc)
- Dynamic query builder (Masterminds/squirrel)
- Testing (ory/dockertest)



Pentagon by Freepik (https://www.flaticon.com/authors/freepik)

Example repo

github.com/johanbrandhorst/grpc-postgres (https://github.com/johanbrandhorst/grpc-postgres)

jackc/pgx

- Pure Go Postgres database/sql driver
- Exposes both database/sql and custom API
- Fast
- Extensive Postgres type support

```
connConfig, err := pgx.ParseConfig(os.Getenv("DATABASE_URL"))
if err != nil {
    // handle error
}

connConfig.Logger = myLogger // supports logrus, zap, zerolog, log15

db := stdlib.OpenDB(*connConfig)

db.QueryRow("select * from users where id=$1", userID)
```

Use custom API at any time for speed

```go
// Get a connection from the *sql.DB connection pool
conn, err := db.Conn(context.Background())
// handle error
defer conn.Close()
err = conn.Raw(func(driverConn interface{}) error {
    conn := driverConn.(*stdlib.Conn).Conn() // conn is a *pgx.Conn
    // Do pgx specific stuff with conn
    return nil
})
// handle error
```

```
$ go test -bench=. -run=^$ -v ./users
BenchmarkAddUsers
BenchmarkAddUsers-8       442702              3335 ns/op
BenchmarkAddUser
BenchmarkAddUser-8         11847             95621 ns/op
PASS
ok      github.com/johanbrandhorst/grpc-postgres/users  23.603s
```

Migrations

- Helps set up the initial database schema
- Helps evolve existing data when requirements change
- Helps rolling back unsuccessful changes

Hiking by Egon Elbre (https://twitter.com/egonelbre)

golang-migrate/migrate

- Supports many different sources (filesystem, S3, bindata)
- Supports Postgres, MySQL, CockroachDB and more
- Can be run in-application or standalone

Go Build by Ashley Willis (https://twitter.com/ashleymcnamara)

Writing migrations

- Create migration files

001_initial_setup.up.sql

```
CREATE TABLE IF NOT EXISTS users (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name TEXT NOT NULL,
    age INTEGER NOT NULL,
    create_time TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

001_initial_setup.down.sql

```
DROP TABLE IF EXISTS users CASCADE;
```

Integrating the migrations

- Generate bindata package

```
$ go-bindata -pkg migrations -ignore bindata -nometadata -prefix migrations -o ./migrations/bindata.go ./migrat
```

- Import into application and run at connection time

```
sourceInstance, err := bindata.WithInstance(bindata.Resource(migrations.AssetNames(), migrations.Asset))
if err != nil {
    // handler err
}
targetInstance, err := postgres.WithInstance(db /* *sql.DB */, new(postgres.Config))
if err != nil {
    // handler err
}
m, err := migrate.NewWithInstance("go-bindata", sourceInstance, "postgres", targetInstance)
if err != nil {
    // handler err
}
err = m.Migrate(1) // Or whatever is the current version
```

kyleconroy/sqlc

- Generates idiomatic Go code from SQL
- Built on the real Postgres query parser
- Generation time type checking of queries
- Supports golang-migrate style schema migration definitions

Source: *sqlc.dev* (https://sqlc.dev)

Dynamic query building

- Listing resources
- Reverse order
- Dynamic filtering
- Issue (https://github.com/kyleconroy/sqlc/discussions/364) lists two options:
  - Write out one SQL query for each variant
  - Use SQL CASE statements

Masterminds/squirrel

- Builder pattern query creator
- Supports most of SQL
- Sacrifices some type safety
- Automatically creates positional parameters

```
sb := squirrel.StatementBuilder.PlaceholderFormat(squirrel.Dollar) // Use postgres placeholders
name := "Johan" // External variable
q := sb.Select(
    "*",
).From(
    "users",
).Where(
    squirrel.Eq{"name": name}, // Becomes positional parameter
)
query, args, err := q.ToSql()
if err != nil {
    // handle err
}
db.Query(query, args...) // SELECT * FROM users WHERE name = $1, [Johan]
```

Testing

ory/dockertest

- Dynamically creates containers for testing
- Uses the Docker API directly, no binary dependencies
- Automatically tear down containers after tests finish
- Can wait for container to start up
- Supports advanced use cases like file upload and log extraction
- Runs the same both in CI and in local testing

Source: Dockertest Github (https://github.com/ory/dockertest)

Conclusion

- We've learned about state management
- We've learned about different storage solutions
- We've learned to use pgx, `golang-migrate`, `sqlc`, `squirrel` and `dockertest`
- We've become Postgres rockstars

GopherAcademy by Ashley Willis (https://twitter.com/ashleymcnamara)

Thank you

Johan Brandhorst
Buf
@johanbrandhorst (http://twitter.com/johanbrandhorst)
https://jbrandhorst.com (https://jbrandhorst.com)