



When it all GOes right

Writing applications with Go for PostgreSQL

Senior Consultant/Developer

Pavlo Golub

MAIL

pavlo.golub@cybertec.at

Twitter

@PavloGolub

WEB

www.cybertec-postgresql.com



ABOUT CYBERTEC



Highly specialized,
fast growing
IT company



International Team
(10 countries),
6 locations worldwide



Database- , Data &
Science Services



Owner managed
since 2000



DATABASE PRODUCTS & TOOLS

 CYBERTEC
MIGRATOR

 **CYPEX**

 CYBERTEC
PGEE

 **scalefield**

DATA
MASKING 

 **WAL**
BOUNCER

 **pg_**
TIMETABLE

PL/pgSQL_  **sec**
CYBERTEC

PGWATCH

 **PG** 
SQUEEZE


YAIM

PG  **SHOW**
PLANS

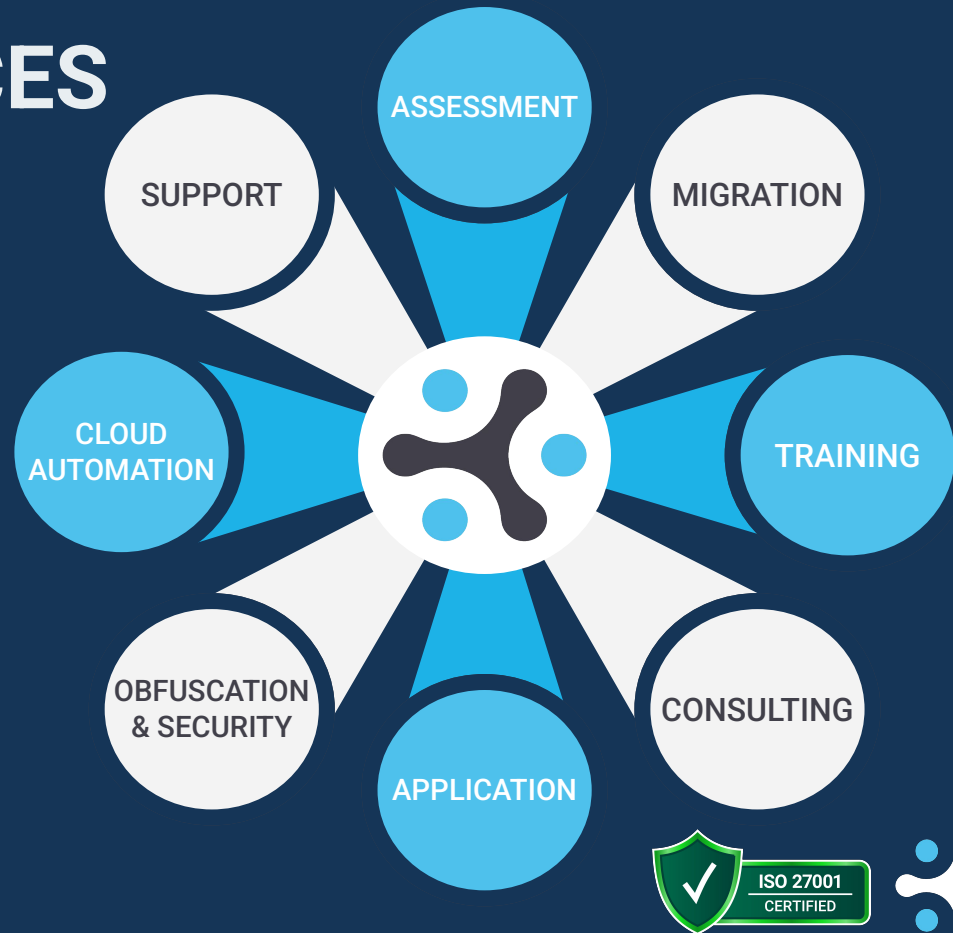
 **POSTGRESQL**
CONFIGURATOR

 **PATRONI**
ENVIRONMENT SETUP



DATABASE SERVICES

- 24/7 Support
- High Availability
- Consulting
- Performance Tuning
- Clustering
- Migration
- Etc.



AUSTRIA (HQ)

CYBERTEC POSTGRESQL
INTERNATIONAL (HQ)

ESTONIA

CYBERTEC POSTGRESQL
NORDIC

SWITZERLAND

CYBERTEC POSTGRESQL SWITZERLAND

POLAND

CYBERTEC POSTGRESQL
POLAND

URUGUAY

CYBERTEC POSTGRESQL
SOUTH AMERICA

SOUTH AFRICA

CYBERTEC POSTGRESQL
SOUTH AFRICA





OpenSource DB + CYBERTEC <https://www.cybertec-postgresql.com/en/>

Our strategic and exclusive partnership with OpenSource DB brings together deep community expertise and cutting-edge PostgreSQL solutions, reinforcing our commitment to delivering comprehensive, end-to-end database services.

OpenSource DB provides:

- Expertise in PostgreSQL Consulting and Professional Services
- Comprehensive Training and Workshops
- Customized Database Solutions
- Global Reach and Experience
- Focus on Open-source



+





Modern **database scaling platform** and **SQL query caching** system.

Boost performance, optimize infrastructure costs, and improve engineering velocity with proactive enhancements for PostgreSQL & MySQL

Reduce hosting costs **by up to 70%**

In memory query read **cache - sub millisecond**

Auto synced to source db changes **in milliseconds**

No **TTLs**, no **cache invalidations**



Available through  **aws** marketplace and self-hosted



AGENDA

- Intro to Go
- IDE's and tools
- Drivers
- Useful extensions
- Testing
- CI/CD

When it all GOes right



Intro to Go

Why Go?

- Native binaries

Go compiles quickly and produces binary code fast.

- Simplicity

It's designed to be simple and easy to learn for developers of all levels.

- Concurrency

Go provides simple and efficient support for concurrent programming.

- Comprehensive tools

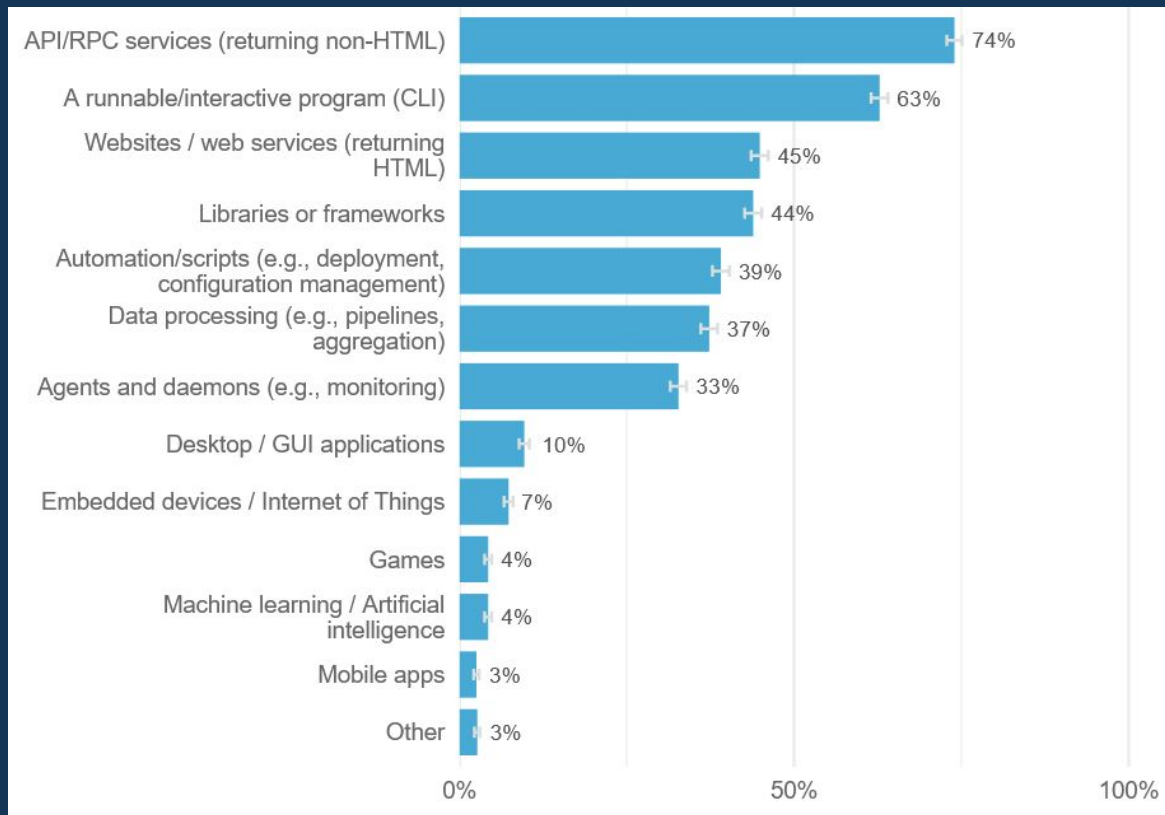
Go comes with a comprehensive set of tools and libraries.

- Cross-platform

Go is designed to allow programs to be compiled and run on any platform.

Go Developer Survey 2024

How did survey respondents use Go?

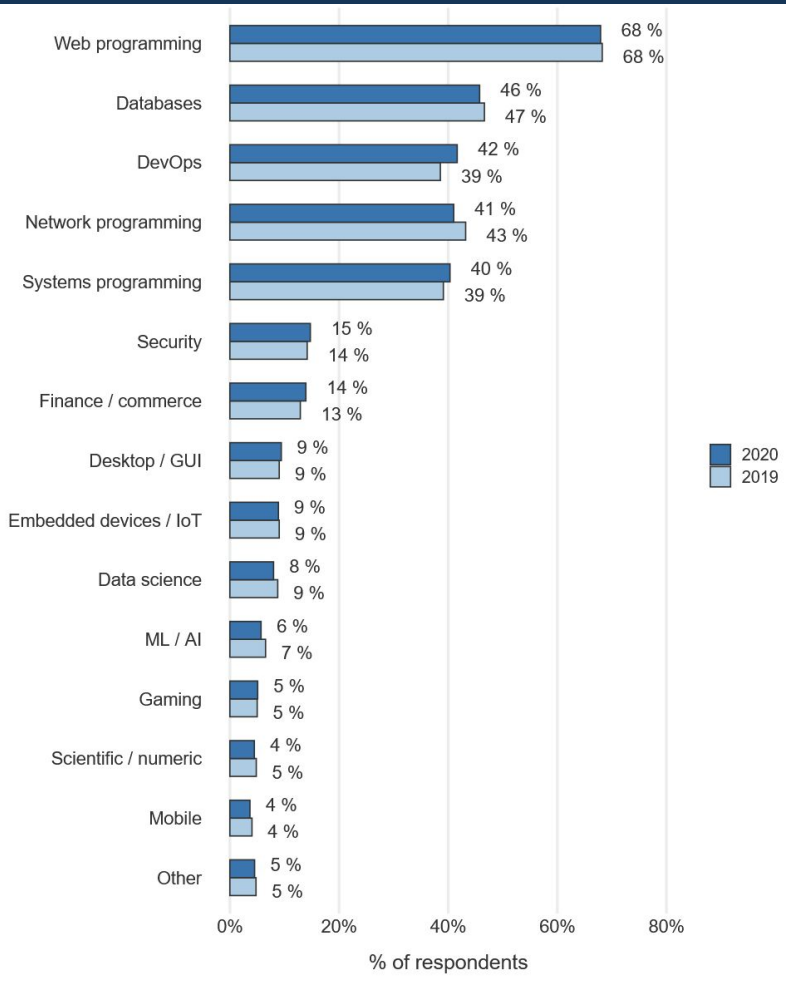


<https://go.dev/blog/survey2024-h1-results>



Go Developer Survey 2020

I work with Go
in the areas listed



● Top products written in Go

- **Kubernetes (K8s)** - production-grade container management
- **Prometheus** - monitoring system and time series database
- **OpenShift** - a family of containerization software products
- **Moby** - a collaborative project for the container ecosystem
- **Hugo** - the world's fastest framework for building websites
- **Grafana** - observability and data visualization platform
- **frp** - a fast reverse proxy to expose a local server behind a NAT
- **Gogs** - painless self-hosted Git service
- **Etcd** - distributed reliable key-value store
- **Caddy** - fast, multi-platform web server with automatic HTTPS

Postgres-related Go products

- **CockroachDB** - a cloud-native distributed SQL database
- **Terraform** - infrastructure managing tool
- **pgweb** - a cross-platform client for PostgreSQL databases
- **stolon** - a cloud native PostgreSQL manager
- **postgres operators**
 - by Zalando, Cybertec, Crunchy, CloudNativePG
- **wal-g** - archives and restoration for Postgres
- **pgwatch** - PostgreSQL metrics monitor/dashboard
- **pg_timetable** - advanced scheduling for PostgreSQL
- **pg_flame** - a flamegraph generator for EXPLAIN output

Top products written in Go

FIND MORE



<https://awesome-go.com/>

<https://github.com/avelino/awesome-go/>

<https://github.com/trending/go/>

When it all GOes right

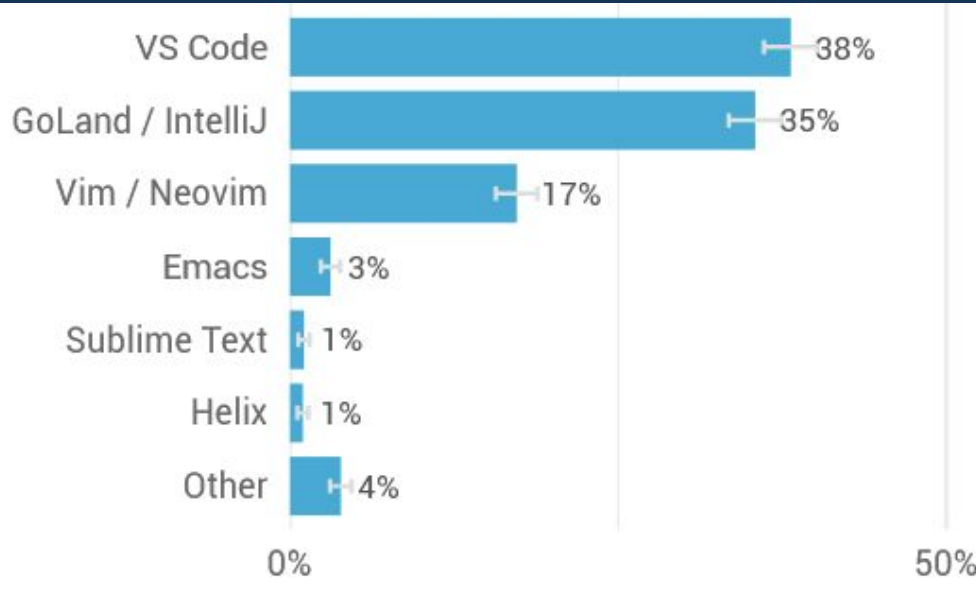


IDE's and tools



Go Developer Survey 2024

“My preferred editor for
Go code is...”

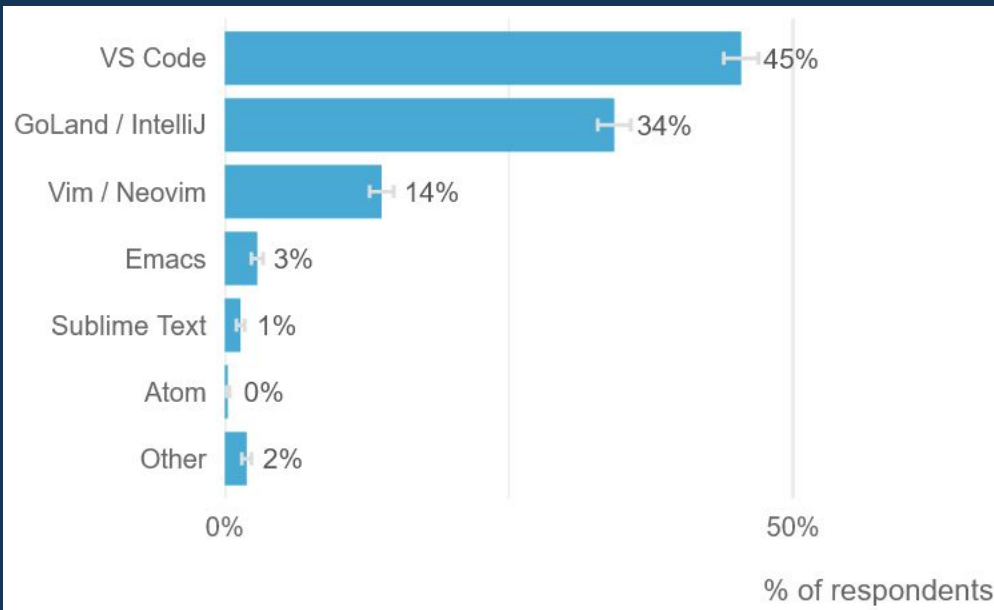


<https://go.dev/blog/survey2024-h2-results>



Go Developer Survey 2022

“My preferred editor for
Go code is...”



<https://go.dev/blog/survey2022-q2-results>

● My environment

- **VSCode** with the official `vscode-go` plugin
- **mfridman/tparse** - summarizes go test output with color tables
- **golangci-lint** - fast linters Runner for Go
- **GitHub Copilot** - the AI developer tool
- **GoReleaser** - deliver Go binaries as quickly and easily as possible
- **PostgreSQL** - most advanced object-relational database
- **Docker** - container virtualization platform

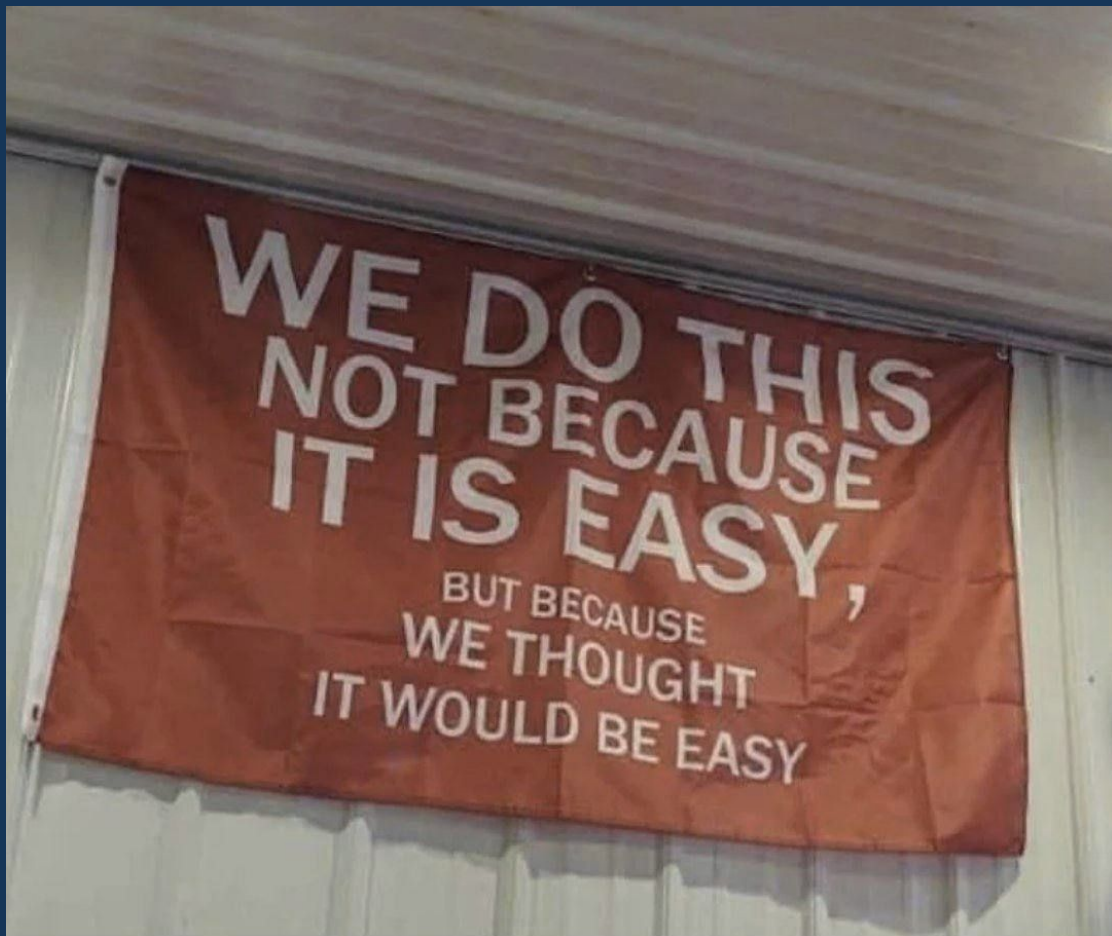
When it all GOes right



PostgreSQL Drivers



Why do people
use ORM's?



Why shouldn't
people
use ORM's?



Pavlo Golub
@PavloGolub



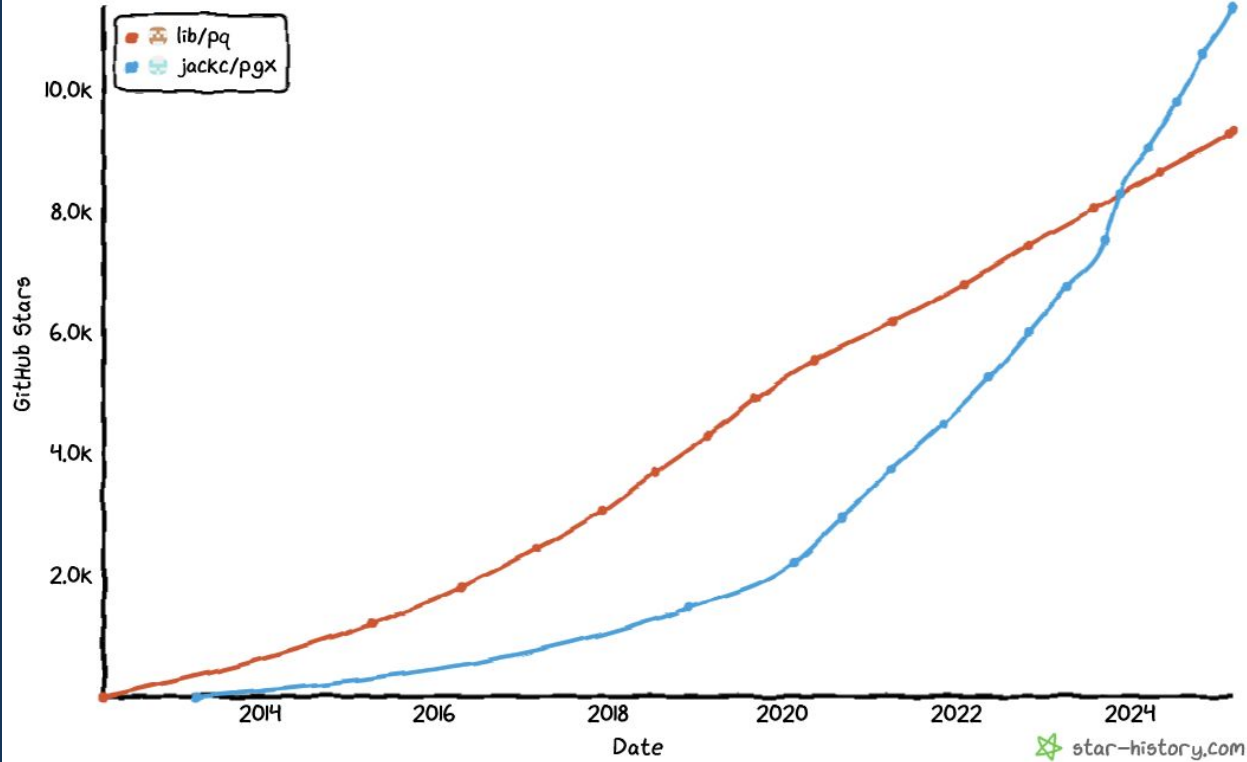
If God wanted us to use ORMs,
God would not have created SQL!

4:59 PM · Dec 14, 2022

● Drivers' availability

- **database/sql** is a set of database access interfaces
 - standard de facto creating database applications
 - needs implementation to work
 - read **go-database-sql.org** for more information
- **github.com/lib/pq** - pure Go Postgres driver for **database/sql**
 - is in maintenance mode for several years
- **github.com/jackc/pgx** — PostgreSQL driver and toolkit for Go
 - is the default choice nowadays
 - use this even with **database/sql**

Star History



pgx vs database/sql

- Use **jackc/pgx** interface (not **database/sql**) when
 - The application only targets PostgreSQL.
 - No other libraries that require **database/sql** are in use.
- Otherwise use **database/sql** with **jackc/pgx/stdlib**
 - compatibility with non-PostgreSQL databases is required
 - when using other libraries that require **database/sql** such as **sqlx** or **gorm**

pgx unique features

- Native Postgres types support
- Automatic statement preparation and caching
- Batch queries
- Single-round trip query mode
- Full TLS connection control
- Binary format support for custom types
- COPY protocol support for faster bulk data loads
- Extendable tracing and logging support
- Connection pool with after-connect hook

pgx unique features

- Listen / Notify
- Conversion of PostgreSQL arrays to Go slice mappings
- `inet`, `cidr`, `hstore`, `json` and `jsonb` native support
- Large object support
- `NULL` mapping to `Null*` struct or pointer to pointer
- Supports `database/sql.Scanner` and `database/sql/driver.Valuer` interfaces for custom types
- Notice response handling
- Simulated nested transactions with savepoints

Hello World: database/sql

```
package main

import (
    "database/sql"
    "fmt"
    "os"

    _ "github.com/jackc/pgx/v5/stdlib"
)

func main() {
    db, err := sql.Open("pgx", os.Getenv("DATABASE_URL"))
    if err != nil {
        fmt.Fprintf(os.Stderr, "Unable to connect to database: %v\n", err)
        os.Exit(1)
    }
    defer db.Close()

    var greeting string
    err = db.QueryRow("select 'Hello, world!'").Scan(&greeting)
    if err != nil {
        fmt.Fprintf(os.Stderr, "QueryRow failed: %v\n", err)
        os.Exit(1)
    }

    fmt.Println(greeting)
}
```

Hello World:

jackc/pgx

```
package main

import (
    "context"
    "fmt"
    "os"

    "github.com/jackc/pgx/v5"
)

func main() {
    conn, err := pgx.Connect(context.Background(), os.Getenv("DATABASE_URL"))
    if err != nil {
        fmt.Fprintf(os.Stderr, "Unable to connect to database: %v\n", err)
        os.Exit(1)
    }
    defer conn.Close(context.Background())

    var greeting string
    err = conn.QueryRow(context.Background(), "select 'Hello, world!'").Scan(&greeting)
    if err != nil {
        fmt.Fprintf(os.Stderr, "QueryRow failed: %v\n", err)
        os.Exit(1)
    }

    fmt.Println(greeting)
}
```

Hello World: pgx/pgxpool

```
package main

import (
    "context"
    "fmt"
    "os"

    "github.com/jackc/pgx/v5/pgxpool"
)

func main() {
    dbpool, err := pgxpool.New(context.Background(), os.Getenv("DATABASE_URL"))
    if err != nil {
        fmt.Fprintf(os.Stderr, "Unable to create connection pool: %v\n", err)
        os.Exit(1)
    }
    defer dbpool.Close()

    var greeting string
    err = dbpool.QueryRow(context.Background(), "select 'Hello, world!'").Scan(&greeting)
    if err != nil {
        fmt.Fprintf(os.Stderr, "QueryRow failed: %v\n", err)
        os.Exit(1)
    }

    fmt.Println(greeting)
}
```

When it all GOes right



Useful extensions



Useful Extensions: database/sql + jmoiron/sqlx

These extensions to the built-in verbs:

- `MustExec() sql.Result` -- Exec, but panic on error
- `Queryx(...)` (`*sqlx.Rows`, `error`) - Query, but return an `sqlx.Rows`
- `QueryRowx(...)` `*sqlx.Row` -- QueryRow, but return an `sqlx.Row`

And these new semantics:

- `Get(dest interface{}, ...)` error
- `Select(dest interface{}, ...)` error



Useful Extensions: database/sql + jmoiron/sqlx

```
type Place struct {  
    Country      string  
    City         sql.NullString  
    TelephoneCode int `db:"telcode"`  
}  
  
rows, err := db.Queryx("SELECT * FROM place")  
for rows.Next() {  
    var p Place  
    err = rows.StructScan(&p)  
}
```

Useful Extensions: database/sql + jmoiron/sqlx

```
p := Place{}
pp := []Place{}

// this will pull the first place directly into p
err = db.Get(&p, "SELECT * FROM place LIMIT 1")

// this will pull places with telcode > 50 into the slice pp
err = db.Select(&pp, "SELECT * FROM place WHERE telcode > ?", 50)

// they work with regular types as well
var id int
err = db.Get(&id, "SELECT count(*) FROM place")

// fetch at most 10 place names
var names []string
err = db.Select(&names, "SELECT name FROM place LIMIT 10")
```




Useful Extensions:
jackc/pgx

It's perfect!
No extensions
needed!

Useful Extensions: jackc/pgx

```
type product struct {
    ID    int32
    Name  string
    Price int32
}

rows, _ := conn.Query(ctx, "select * from products where price < $1", 12)
products, err := pgx.CollectRows(rows, pgx.RowToStructByName[product])
if err != nil {
    fmt.Printf("CollectRows error: %v", err)
    return
}

for _, p := range products {
    fmt.Printf("%s: %d\n", p.Name, p.Price)
}
```



Useful Extensions:

sqlc-dev/sqlc

sqlc generates **type-safe** code from SQL. Here's how it works:

1. You write queries in SQL.
2. You run sqlc to generate code with type-safe interfaces to those queries.
3. You write application code that calls the generated code.

Check out an [interactive example](#) to see it in action

When it all GOes right



Testing

● Testing Approaches

- Real PostgreSQL server
 - running locally
 - **testcontainers/testcontainers-go**
- Mocking libraries
 - **DATA-DOG/go-sqlmock**
 - **pashagolub/pgxmock**
- Mock PostgreSQL wire protocol
 - **jackc/pgmock**
 - **cockroachdb/cockroach-go/testserver**
 - **cockroachdb/copyist**

Mocking Example: pgxmock

```
type PgxFace interface {
    Begin(context.Context) (pgx.Tx, error)
    Close(context.Context) error
}

func recordStats(db PgxFace, userID, productID int) (err error) {
    if tx, err := db.Begin(context.Background()); err != nil {
        return
    }
    defer func() {
        switch err {
        case nil:
            err = tx.Commit(context.Background())
        default:
            _ = tx.Rollback(context.Background())
        }
    }()
    sql := "UPDATE products SET views = views + 1"
    if _, err = tx.Exec(context.Background(), sql); err != nil {
        return
    }
    sql = "INSERT INTO product_viewers (user_id, product_id) VALUES ($1, $2)"
    if _, err = tx.Exec(context.Background(), sql, userID, productID); err != nil {
        return
    }
    return
}
```

Mocking Example: pgxmock

```
// a successful case
func TestShouldUpdateStats(t *testing.T) {
    mock, err := pgxmock.NewPool()
    if err != nil {
        t.Fatal(err)
    }
    defer mock.Close()

    mock.ExpectBegin()
    mock.ExpectExec("UPDATE products").
        WillReturnResult(pgxmock.NewResult("UPDATE", 1))
    mock.ExpectExec("INSERT INTO product_viewers").
        WithArgs(2, 3).
        WillReturnResult(pgxmock.NewResult("INSERT", 1))
    mock.ExpectCommit()

    // now we execute our method
    if err = recordStats(mock, 2, 3); err != nil {
        t.Errorf("error was not expected while updating: %s", err)
    }

    // we make sure that all expectations were met
    if err := mock.ExpectationsWereMet(); err != nil {
        t.Errorf("there were unfulfilled expectations: %s", err)
    }
}
```

Mocking Example: pgxmock

```
// a failing test case
func TestShouldRollbackStatUpdatesOnFailure(t *testing.T) {
    mock, err := pgxmock.NewPool()
    if err != nil {
        t.Fatal(err)
    }
    defer mock.Close()

    mock.ExpectBegin()
    mock.ExpectExec("UPDATE products").
        WillReturnResult(pgxmock.NewResult("UPDATE", 1))
    mock.ExpectExec("INSERT INTO product_viewers").
        WithArgs(2, 3).
        WillReturnError(fmt.Errorf("some error"))
    mock.ExpectRollback()

    // now we execute our method
    if err = recordStats(mock, 2, 3); err == nil {
        t.Errorf("was expecting an error, but there was none")
    }

    // we make sure that all expectations were met
    if err := mock.ExpectationsWereMet(); err != nil {
        t.Errorf("there were unfulfilled expectations: %s", err)
    }
}
```


Coverage:

go test + tparse

```
basic master !2 ~2 1 1.19.5 go test ./... -coverprofile='c.out' -json | tparse -all
```

STATUS	ELAPSED	TEST	PACKAGE
PASS	0.00	TestShouldUpdateStats	basic
PASS	0.00	TestShouldRollbackStatUpdatesOnFailure	basic

STATUS	ELAPSED	PACKAGE	COVER	PASS	FAIL	SKIP
PASS	0.30s	github.com/pashagolub/pgxmock/v2/examples/basic	60.0%	2	0	0

```
basic master !2 ~2 1 1.19.5 go tool cover -func='c.out'
```

github.com/pashagolub/pgxmock/v2/examples/basic/basic.go:15:	recordStats	85.7%
github.com/pashagolub/pgxmock/v2/examples/basic/basic.go:39:	main	0.0%
total:	(statements)	60.0%

```
basic master !2 ~2 1 1.19.5 go tool cover -html='c.out'
```

Coverage:

go tool cover

```
github.com/pashagolub/pgxmock/v2/examples/basic/basic.go (60.0%) ▾ not tracked not covered covered

package main

import (
    "context"

    pgx "github.com/jackc/pgx/v5"
    pgxpool "github.com/jackc/pgx/v5/pgxpool"
)

type PgxFace interface {
    Begin(context.Context) (pgx.Tx, error)
    Close()
}

func recordStats(db PgxFace, userID, productID int) (err error) {
    tx, err := db.Begin(context.Background())
    if err != nil {
        return
    }
    defer func() {
        switch err {
        case nil:
            err = tx.Commit(context.Background())
        default:
            _ = tx.Rollback(context.Background())
        }
    }()
    sql := "UPDATE products SET views = views + 1"
    if _, err = tx.Exec(context.Background(), sql); err != nil {
        return
    }
    sql = "INSERT INTO product_viewers (user_id, product_id) VALUES ($1, $2)"
    if _, err = tx.Exec(context.Background(), sql, userID, productID); err != nil {
        return
    }
    return
}

func main() {
    // @NOTE: the real connection is not required for tests
    db, err := pgxpool.New(context.Background(), "postgres://rolname@hostname/dbname")
    if err != nil {
        panic(err)
    }
    defer db.Close()

    if err = recordStats(db, 1 /*some user id*/, 5 /*some product id*/); err != nil {
        panic(err)
    }
}
```

When it all GOes right



CI/CD



GitHub Integration

- **Dependabot** - maintains repository's dependencies automatically
- **CodeQL** - action runs analysis engine to find security vulnerabilities
- **Build & Test** - action runs on each pull request, or manually
- **Release** - action runs on new tag, publishes release
- **Docker** - action runs on every commit, publishes Docker images

Github Actions: Dependabot

```
1  version: 2
2  updates:
3
4    # Maintain dependencies for Go modules
5    - package-ecosystem: gomod
6      directory: "/"
7      schedule:
8        interval: daily
9        time: "04:00"
10     open-pull-requests-limit: 10
11
12    # Maintain dependencies for GitHub Actions
13    - package-ecosystem: "github-actions"
14      directory: "/"
15      schedule:
16        interval: "daily"
```



Github Actions: CodeQL

```
1 name: "CodeQL"
2
3 on:
4   push:
5     branches: [ master ]
6   pull_request:
7     branches: [ master ]
8   schedule:
9     - cron: '19 11 * * 6'
10
11 jobs:
12   analyze:
13     name: Analyze
14     runs-on: ubuntu-latest
15     permissions:
16       actions: read
17       contents: read
18       security-events: write
19
20     strategy:
21       fail-fast: false
22       matrix:
23         language: [ 'go' ]
24
25     steps:
26     - name: Checkout repository
27       uses: actions/checkout@v3
28
29     # Initializes the CodeQL tools for scanning.
30     - name: Initialize CodeQL
31       uses: github/codeql-action/init@v2
32       with:
33         languages: ${{ matrix.language }}
34
35     - name: Autobuild
36       uses: github/codeql-action/autobuild@v2
37
38     - name: Perform CodeQL Analysis
39       uses: github/codeql-action/analyze@v2
```

Github Actions:

Build & Test

```
90 test-postgresql-ubuntu:
91   if: true # false to skip job during debug
92   name: Test and Build on Ubuntu
93   runs-on: ubuntu-latest
94   steps:
95
96     - name: Start PostgreSQL on Ubuntu
97       run: |
98         sudo systemctl start postgresql.service
99         pg_isready
100
101     - name: Create scheduler user
102       run: |
103         sudo -u postgres psql --command="CREATE USER scheduler PASSWORD 'somestrong'" --command="\du"
104
105     - name: Create timetable database
106       run: |
107         sudo -u postgres createdb --owner=scheduler timetable
108         PGPASSWORD=somestrong psql --username=scheduler --host=localhost --list timetable
109
110     - name: Check out code
111       uses: actions/checkout@v3
112
113     - name: Set up Golang
114       uses: actions/setup-go@v3
115       with:
116         go-version: '1.19'
117
118     - name: Get dependencies
119       run: |
120         go mod download
121         go version
122
123     - name: GolangCI-Lint
124       uses: golangci/golangci-lint-action@v3
125       with:
126         version: latest
127
128     - name: Test
129       run: go test -failfast -v -timeout=300s -p 1 -coverprofile=profile.cov ./...
130
131     - name: Coveralls
132       uses: shogo82148/actions-goveralls@v1
133       with:
134         path-to-profile: profile.cov
```



Github Actions: Release

```
1  name: Release
2  on:
3    release:
4      types: [created]
5
6  jobs:
7
8    goreleaser:
9      if: true # false to skip job during debug
10     runs-on: ubuntu-latest
11     name: goreleaser
12     steps:
13
14       - name: Set up Golang
15         uses: actions/setup-go@v3
16         with:
17           go-version: '1.19'
18
19       - name: Check out code into the Go module directory
20         uses: actions/checkout@v3
21
22       - name: Unshallow
23         run: git fetch --prune --unshallow
24
25       - name: Release via goreleaser
26         uses: goreleaser/goreleaser-action@v4
27         with:
28           args: release
29         env:
30           GITHUB_TOKEN: ${ secrets.GITHUB_TOKEN }
```


Github Actions:

Docker

```
16 docker:
17   if: true # false to skip job during debug
18   runs-on: ubuntu-latest
19   steps:
20
21   - name: Check out code
22     uses: actions/checkout@v3
23
24   - name: Set up Golang
25     uses: actions/setup-go@v3
26     with:
27       go-version: '1.19'
28
29   # despite the fact docker will build binary internally
30   # we want to stop workflow in case of any error before pushing to registry
31   - name: Get dependencies and Build
32     run: |
33       go version
34       go mod download
35       go build
36
37   - name: Version strings
38     id: version
39     run: |
40       echo "RELEASE_TIME=$(git show -s --format=%cI HEAD)" >> $GITHUB_OUTPUT
41
42   - name: Publish to Registry
43     uses: elgohr/Publish-Docker-Github-Action@v5
44     env:
45       VERSION: ${ github.ref_name }
46       COMMIT: ${ github.sha }
47       DATE: ${ steps.version.outputs.RELEASE_TIME }
48     with:
49       name: cybertecpostgresql/pg_timetable
50       username: ${ secrets.DOCKER_USERNAME }
51       password: ${ secrets.DOCKER_PASSWORD }
52       buildargs: VERSION,COMMIT,DATE
```

When it all GOes right



Takeaways

Takeaways

- Go language is popular, fast, easy and well-scaled
- 45% of developers use Go to work with databases (go.dev survey)
- You can use your preferred editor or use special IDE's to work
- Kubernetes operators (including Postgres ones) are written in Go
- Go is flexible when working with Postgres: use **sql** or **pgx** interfaces
- Go has powerful programming tools and GitHub/GitLab integration
- Go is backwards-compatible. The API's may grow but not in a way that breaks existing Go v1 code



Be Inspired

“I would rather have questions that can't be answered than answers that can't be questioned.”

Richard Feynman

DON'T BE A STRANGER



PERSONAL GITHUB

pashagolub.github.io



CYBERTEC BLOG

www.cybertec-postgresql.com/en/blog/



CYBERTEC GITHUB

www.github.com/cybertec-postgresql

#StandWithUkraine

