Go Database Tools Comparison: db/sql, GORM, sqlx, and sqlc

Core Differences

Library	Type	SQL Generation	Type Safety	Learning Curve	Best For
database/sql	Standard library	Manual SQL	Limited	Moderate	Simple projects, full control
GORM	Full ORM	Generated	Partial	Steeper	Rapid development, model-based apps
sqlx	Lightweight wrapper	Manual SQL	Improved	Low	Balance of control and convenience
sqlc	Code generator	Manual SQL → Generated code	Excellent	Low	Type safety, performance

database/sql (Standard Library)

Strengths:

- No dependencies
- Complete control over SQL
- Good performance

Weaknesses:

- Verbose code
- Manual mapping of rows to structs
- Error-prone

Basic usage:

go

```go
import (
    "database/sql"
    _ "github.com/lib/pq" // PostgreSQL driver
)

func main() {
    db, err := sql.Open("postgres", "postgres://user:password@localhost/dbname")
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()

    rows, err := db.Query("SELECT id, name FROM users WHERE age > $1", 18)
    if err != nil {
        log.Fatal(err)
    }
    defer rows.Close()

    var users []User
    for rows.Next() {
        var u User
        if err := rows.Scan(&u.ID, &u.Name); err != nil {
            log.Fatal(err)
        }
        users = append(users, u)
    }
}
```

GORM

Strengths:

- Quick development
- Handles migrations
- Many built-in features (hooks, associations)

Weaknesses:

- Performance overhead
- "Magic" behavior

- SQL abstraction can lead to inefficient queries

Basic usage:

go

```go
import (
    "gorm.io/gorm"
    "gorm.io/driver/postgres"
)

type User struct {
    gorm.Model
    Name  string
    Age   int
    Posts []Post
}

type Post struct {
    gorm.Model
    Title   string
    Content string
    UserID  uint
}

func main() {
    dsn := "postgres://user:password@localhost/dbname"
    db, err := gorm.Open(postgres.Open(dsn), &gorm.Config{})
    if err != nil {
        log.Fatal(err)
    }

    // Auto-migration
    db.AutoMigrate(&User{}, &Post{})

    // Create
    db.Create(&User{Name: "John", Age: 25})

    // Read
    var user User
    db.First(&user, "name = ?", "John")

    // Update
    db.Model(&user).Update("Age", 26)

    // Delete
    db.Delete(&user)

    // Association
```

```
    var userWithPosts User
    db.Preload("Posts").First(&userWithPosts, 1)
}
```

sqlx

Strengths:

- Simpler than raw database/sql
- Minimal overhead
- Maintains SQL control
- Struct field mapping

Weaknesses:

- No migration tools
- No query generation

Basic usage:

go

```go
import (
    "github.com/jmoiron/sqlx"
    _ "github.com/lib/pq"
)

type User struct {
    ID   int    `db:"id"`
    Name string `db:"name"`
    Age  int    `db:"age"`
}

func main() {
    db, err := sqlx.Connect("postgres", "postgres://user:password@localhost/dbname")
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()

    // Query and scan into struct
    users := []User{}
    err = db.Select(&users, "SELECT id, name, age FROM users WHERE age > $1", 18)
    if err != nil {
        log.Fatal(err)
    }

    // Single row
    var user User
    err = db.Get(&user, "SELECT id, name, age FROM users WHERE id = $1", 1)

    // Named parameters
    _, err = db.NamedExec(
        "INSERT INTO users (name, age) VALUES (:name, :age)",
        map[string]interface{}{"name": "Jane", "age": 30},
    )
}
```

sqlc

Strengths:

- Type-safe generated code

- Excellent performance
- SQL-first approach
- IDE assistance

Weaknesses:

- Requires code generation step
- No migration tools

Basic usage:

1. Define SQL queries in .sql files:

```sql
-- query.sql
-- name: GetUser :one
SELECT id, name, age FROM users
WHERE id = $1;

-- name: ListUsers :many
SELECT id, name, age FROM users
WHERE age > $1
ORDER BY name;

-- name: CreateUser :one
INSERT INTO users (name, age)
VALUES ($1, $2)
RETURNING id, name, age;
```

2. Generate Go code:

```bash
sqlc generate
```

3. Use generated code:

go

```go
import (
    "context"
    "database/sql"
    "log"

    _ "github.com/lib/pq"
    "github.com/your/project/db"
)

func main() {
    conn, err := sql.Open("postgres", "postgres://user:password@localhost/dbname")
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()

    queries := db.New(conn)
    ctx := context.Background()

    // Create user
    user, err := queries.CreateUser(ctx, db.CreateUserParams{
        Name: "Alice",
        Age:  28,
    })

    // Get user
    user, err = queries.GetUser(ctx, 1)

    // List users
    users, err := queries.ListUsers(ctx, 18)
}
```

When to use each

- **database/sql**: When you need complete control or have very simple needs
- **GORM**: When productivity matters more than performance, for complex models with relationships
- **sqlx**: When you want a balance of control and convenience
- **sqlc**: When you want type safety and high performance while keeping SQL-level control

Migration Approaches

- **database/sql**: No built-in migrations; use external tools like golang-migrate
- **GORM**: Built-in migrations with AutoMigrate
- **sqlx**: No built-in migrations; use external tools
- **sqlc**: No built-in migrations; use external tools

For a robust solution, I'd suggest:

- Simple apps: sqlx
- Complex domain models: GORM
- Performance-critical apps: sqlc
- Maximum control: database/sql