## Introduction

The `database/sql` package provides a generic interface around SQL (or SQL-like) databases. See the official documentation for details.

This page provides example usage patterns.

## Database driver

The `database/sql` package must be used in conjunction with a database driver. See https://go.dev/s/sqldrivers for a list of drivers.

The documentation below assumes a driver has been imported.

## Connecting to a database

`Open` is used to create a database handle:

```
db, err := sql.Open(driver, dataSourceName)
```

Where driver specifies a database driver and dataSourceName specifies database-specific connection information such as database name and authentication credentials.

Note that `Open` does not directly open a database connection: this is deferred until a query is made. To verify that a connection can be made before making a query, use the `PingContext` method:

```
if err := db.PingContext(ctx); err != nil {
  log.Fatal(err)
}
```

After use, the database is closed using `Close`.

## Executing queries

`ExecContext` is used for queries where no rows are returned:

```
result, err := db.ExecContext(ctx,
    "INSERT INTO users (name, age) VALUES ($1, $2)",
    "gopher",
    27,
)
```

Where result contains the last insert ID and number of rows affected. The availability of these values is dependent on the database driver.

`QueryContext` is used for retrieval:

```
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)
    }
    fmt.Printf("%s is %d\n", name, age)
}
if err := rows.Err(); err != nil {
    log.Fatal(err)
}
```

`QueryRowContext` is used where only a single row is expected:

```
var age int64
err := db.QueryRowContext(ctx, "SELECT age FROM users WHERE name = $1", name).Scan(&age)
```

Prepared statements can be created with `PrepareContext`:

```
age := 27
stmt, err := db.PrepareContext(ctx, "SELECT name FROM users WHERE age = $1")
if err != nil {
    log.Fatal(err)
}
rows, err := stmt.Query(age)
// process rows
```

`ExecContext`, `QueryContext` and `QueryRowContext` can be called on statements.
After use, a statement should be closed with `Close`.

## Transactions

Transactions are started with `BeginTx`:

```
tx, err := db.BeginTx(ctx, nil)
if err != nil {
    log.Fatal(err)
}
```

The `ExecContext`, `QueryContext`, `QueryRowContext` and `PrepareContext` methods already covered can be used in a transaction.

A transaction must end with a call to `Commit` or `Rollback`.

## Dealing with NULL

If a database column is nullable, one of the types supporting null values should be passed to Scan.

For example, if the name column in the names table is nullable:

```
var name sql.NullString
err := db.QueryRowContext(ctx, "SELECT name FROM names WHERE id = $1", id).Scan(&name)
...
if name.Valid {
    // use name.String
} else {
    // value is NULL
}
```

Only `NullByte`, `NullBool`, `NullFloat64`, `NullInt64`, `NullInt32` `NullInt16`, `NullString` and `NullTime` are implemented in `database/sql`. Implementations of database-specific null types are left to the database driver. User types supporting `NULL` can be created by implementing interfaces `database/sql/driver.Valuer` and `database/sql.Scanner`.

You can also pass pointer types. Be careful for performance issues as it requires extra memory allocations.

```
var name *string
err := db.QueryRowContext(ctx, "SELECT name FROM names WHERE id = $1", id).Scan(&name)
```