sqlc

Release 1.6.0

Kyle Conroy

Mar 01, 2021

OVERVIEW

1 Installing sqlc **3**
1.1 macOS . . . 3
1.2 Ubuntu . . . 3
1.3 go get . . . 3
1.4 Docker . . . 3
1.5 Downloads . . . 3
1.6 Tip Releases . . . 4

2 Getting started **5**

3 Retrieving rows **9**
3.1 Selecting columns . . . 11
3.2 Passing a slice as a parameter to a query . . . 12

4 Counting rows **15**

5 Inserting rows **17**
5.1 Returning columns from inserted rows . . . 18

6 Updating rows **21**

7 Deleting rows **23**

8 Preparing queries **25**

9 Using transactions **27**

10 Naming parameters **29**

11 Modifying the database schema **31**
11.1 Handling SQL migrations . . . 31

12 Configuring generated structs **35**
12.1 Naming scheme . . . 35
12.2 JSON tags . . . 35

13 CLI **37**

14 Configuration file (version 1) **39**
14.1 Type Overrides . . . 40
14.2 Per-Column Type Overrides . . . 40
14.3 Package Level Overrides . . . 41

14.4 Renaming Struct Fields 41

15 Datatypes **43**
15.1 Arrays 43
15.2 Dates and Time 43
15.3 Enums 44
15.4 Null 44
15.5 UUIDs 45

16 Query annotations **47**
16.1 `:exec` 47
16.2 `:execresult` 47
16.3 `:execrows` 48
16.4 `:many` 48
16.5 `:one` 48

17 Developing sqlc **49**
17.1 Building 49
17.2 Running Tests 49
17.3 Regenerate expected test output 50

18 Privacy and data collection **51**
18.1 Our Privacy Pledge 51
18.2 Distribution Channels 51

And lo, the Great One looked down upon the people and proclaimed: "SQL is actually pretty great"

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

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

Seriously, it's that easy. You don't have to write any boilerplate SQL querying code ever again.

CHAPTER

ONE

INSTALLING SQLC

sqlc is distributed as a single binary with zero dependencies.

1.1 macOS

```
brew install sqlc
```

1.2 Ubuntu

```
sudo snap install sqlc
```

1.3 go get

```
go get github.com/kyleconroy/sqlc/cmd/sqlc
```

1.4 Docker

```
docker pull kjconroy/sqlc
```

Run `sqlc` using `docker run`:

```
docker run --rm -v $(pwd):/src -w /src kjconroy/sqlc generate
```

1.5 Downloads

Binaries for a given release can be downloaded from the stable channel on Equinox or the latest GitHub release.

1.6 Tip Releases

Each commit is deployed to the `devel` channel on Equinox:

- Linux
- macOS
- Windows

CHAPTER

TWO

GETTING STARTED

Okay, enough hype, let's see it in action.

First you pass the following SQL to `sqlc generate`:

```sql
CREATE TABLE authors (
  id   BIGSERIAL PRIMARY KEY,
  name text      NOT NULL,
  bio  text
);

-- name: GetAuthor :one
SELECT * FROM authors
WHERE id = $1 LIMIT 1;

-- name: ListAuthors :many
SELECT * FROM authors
ORDER BY name;

-- name: CreateAuthor :one
INSERT INTO authors (
  name, bio
) VALUES (
  $1, $2
)
RETURNING *;

-- name: DeleteAuthor :exec
DELETE FROM authors
WHERE id = $1;
```

And then in your application code you'd write:

```go
// list all authors
authors, err := db.ListAuthors(ctx)
if err != nil {
    return err
}
fmt.Println(authors)

// create an author
insertedAuthor, err := db.CreateAuthor(ctx, db.CreateAuthorParams{
        Name: "Brian Kernighan",
        Bio:  sql.NullString{String: "Co-author of The C Programming Language and The␣
↪Go Programming Language", Valid: true},
```

(continues on next page)

(continued from previous page)

```
})
if err != nil {
        return err
}
fmt.Println(insertedAuthor)

// get the author we just inserted
fetchedAuthor, err := db.GetAuthor(ctx, insertedAuthor.ID)
if err != nil {
        return err
}
// prints true
fmt.Println(reflect.DeepEqual(insertedAuthor, fetchedAuthor))
```

To make that possible, sqlc generates readable, **idiomatic** Go code that you otherwise would have had to write yourself. Take a look:

```
package db

import (
        "context"
        "database/sql"
)

type Author struct {
        ID   int64
        Name string
        Bio  sql.NullString
}

const createAuthor = `-- name: CreateAuthor :one
INSERT INTO authors (
  name, bio
) VALUES (
  $1, $2
)
RETURNING id, name, bio
`

type CreateAuthorParams struct {
        Name string
        Bio  sql.NullString
}

func (q *Queries) CreateAuthor(ctx context.Context, arg CreateAuthorParams) (Author,␣
↪error) {
        row := q.db.QueryRowContext(ctx, createAuthor, arg.Name, arg.Bio)
        var i Author
        err := row.Scan(&i.ID, &i.Name, &i.Bio)
        return i, err
}

const deleteAuthor = `-- name: DeleteAuthor :exec
DELETE FROM authors
WHERE id = $1
`
```

(continues on next page)

(continued from previous page)

```go
func (q *Queries) DeleteAuthor(ctx context.Context, id int64) error {
	_, err := q.db.ExecContext(ctx, deleteAuthor, id)
	return err
}

const getAuthor = `-- name: GetAuthor :one
SELECT id, name, bio FROM authors
WHERE id = $1 LIMIT 1
`

func (q *Queries) GetAuthor(ctx context.Context, id int64) (Author, error) {
	row := q.db.QueryRowContext(ctx, getAuthor, id)
	var i Author
	err := row.Scan(&i.ID, &i.Name, &i.Bio)
	return i, err
}

const listAuthors = `-- name: ListAuthors :many
SELECT id, name, bio FROM authors
ORDER BY name
`

func (q *Queries) ListAuthors(ctx context.Context) ([]Author, error) {
	rows, err := q.db.QueryContext(ctx, listAuthors)
	 if err != nil {
		return nil, err
	}
	defer rows.Close()
	var items []Author
	for rows.Next() {
		var i Author
		if err := rows.Scan(&i.ID, &i.Name, &i.Bio); err != nil {
			return nil, err
		}
		items = append(items, i)
	}
	if err := rows.Close(); err != nil {
		return nil, err
	}
	if err := rows.Err(); err != nil {
		return nil, err
	}
	return items, nil
}

type DBTX interface {
	ExecContext(context.Context, string, ...interface{}) (sql.Result, error)
	PrepareContext(context.Context, string) (*sql.Stmt, error)
	QueryContext(context.Context, string, ...interface{}) (*sql.Rows, error)
	QueryRowContext(context.Context, string, ...interface{}) *sql.Row
}

func New(db DBTX) *Queries {
	return &Queries{db: db}
}
```

(continues on next page)

(continued from previous page)

```go
type Queries struct {
	db DBTX
}

func (q *Queries) WithTx(tx *sql.Tx) *Queries {
	return &Queries{
		db: tx,
	}
}
```

CHAPTER

THREE

RETRIEVING ROWS

To generate a database access method, annotate a query with a specific comment.

```
CREATE TABLE authors (
  id          SERIAL PRIMARY KEY,
  bio         text   NOT NULL,
  birth_year  int    NOT NULL
);


-- name: GetAuthor :one
SELECT * FROM authors
WHERE id = $1;

-- name: ListAuthors :many
SELECT * FROM authors
ORDER BY id;
```

A few new pieces of code are generated beyond the `Author` struct. An interface for the underlying database is generated. The `*sql.DB` and `*sql.Tx` types satisty this interface.

The database access methods are added to a `Queries` struct, which is created using the `New` method.

Note that the `*` in our query has been replaced with explicit column names. This change ensures that the query will never return unexpected data.

Our query was annotated with `:one`, meaning that it should only return a single row. We scan the data from that one into a `Author` struct.

Since the get query has a single parameter, the `GetAuthor` method takes a single `int` as an argument.

Since the list query has no parameters, the `ListAuthors` method accepts no arguments.

```
package db

import (
        "context"
        "database/sql"
)

type Author struct {
        ID        int
        Bio       string
        BirthYear int
}

```

(continues on next page)

(continued from previous page)

```
type DBTX interface {
	QueryContext(context.Context, string, ...interface{}) (*sql.Rows, error)
	QueryRowContext(context.Context, string, ...interface{}) *sql.Row
}

func New(db DBTX) *Queries {
	return &Queries{db: db}
}

type Queries struct {
	db DBTX
}

const getAuthor = `-- name: GetAuthor :one
SELECT id, bio, birth_year FROM authors
WHERE id = $1
`

func (q *Queries) GetAuthor(ctx context.Context, id int) (Author, error) {
  row := q.db.QueryRowContext(ctx, getAuthor, id)
  var i Author
  err := row.Scan(&i.ID, &i.Bio, &i.BirthYear)
  return i, err
}

const listAuthors = `-- name: ListAuthors :many
SELECT id, bio, birth_year FROM authors
ORDER BY id
`

func (q *Queries) ListAuthors(ctx context.Context) ([]Author, error) {
  rows, err := q.db.QueryContext(ctx, listAuthors)
  if err != nil {
    return nil, err
  }
  defer rows.Close()
  var items []Author
  for rows.Next() {
    var i Author
    if err := rows.Scan(&i.ID, &i.Bio, &i.BirthYear); err != nil {
      return nil, err
    }
    items = append(items, i)
  }
  if err := rows.Close(); err != nil {
    return nil, err
  }
  if err := rows.Err(); err != nil {
    return nil, err
  }
  return items, nil
}
```

3.1 Selecting columns

```
CREATE TABLE authors (
  id         SERIAL PRIMARY KEY,
  bio        text   NOT NULL,
  birth_year int    NOT NULL
);

-- name: GetBioForAuthor :one
SELECT bio FROM authors
WHERE id = $1;

-- name: GetInfoForAuthor :one
SELECT bio, birth_year FROM authors
WHERE id = $1;
```

When selecting a single column, only that value that returned. The `GetBioForAuthor` method takes a single `int` as an argument and returns a `string` and an `error`.

When selecting multiple columns, a row record (method-specific struct) is returned. In this case, `GetInfoForAuthor` returns a struct with two fields: `Bio` and `BirthYear`.

```go
package db

import (
	"context"
	"database/sql"
)

type DBTX interface {
	QueryRowContext(context.Context, string, ...interface{}) *sql.Row
}

func New(db DBTX) *Queries {
	return &Queries{db: db}
}

type Queries struct {
	db DBTX
}

const getBioForAuthor = `-- name: GetBioForAuthor :one
SELECT bio FROM authors
WHERE id = $1
`

func (q *Queries) GetBioForAuthor(ctx context.Context, id int) (string, error) {
	row := q.db.QueryRowContext(ctx, getBioForAuthor, id)
	var i string
	err := row.Scan(&i)
	return i, err
}

const getInfoForAuthor = `-- name: GetInfoForAuthor :one
SELECT bio, birth_year FROM authors
WHERE id = $1
`
```

(continues on next page)

(continued from previous page)

```
type GetInfoForAuthorRow struct {
        Bio       string
        BirthYear int
}

func (q *Queries) GetBioForAuthor(ctx context.Context, id int) (GetBioForAuthor,␣
↪error) {
        row := q.db.QueryRowContext(ctx, getInfoForAuthor, id)
        var i GetBioForAuthor
        err := row.Scan(&i.Bio, &i.BirthYear)
        return i, err
}
```

3.2 Passing a slice as a parameter to a query

In PostgreSQL, ANY allows you to check if a value exists in an array expression. Queries using ANY with a single parameter will generate method signatures with slices as arguments.

```
CREATE TABLE authors (
  id          SERIAL PRIMARY KEY,
  bio         text   NOT NULL,
  birth_year int     NOT NULL
);

-- name: ListAuthorsByIDs :many
SELECT * FROM authors
WHERE id = ANY($1::int[]);
```

The above SQL will generate the following code:

```
package db

import (
        "context"
        "database/sql"

  "github.com/lib/pq"
)

type Author struct {
        ID        int
        Bio       string
        BirthYear int
}

type DBTX interface {
        QueryContext(context.Context, string, ...interface{}) (*sql.Rows, error)
        QueryRowContext(context.Context, string, ...interface{}) *sql.Row
}

func New(db DBTX) *Queries {
        return &Queries{db: db}
}
```

(continues on next page)

(continued from previous page)

```
type Queries struct {
        db DBTX
}

const listAuthors = `-- name: ListAuthorsByIDs :many
SELECT id, bio, birth_year FROM authors
WHERE id = ANY($1::int[])
`

func (q *Queries) ListAuthorsByIDs(ctx context.Context, ids []int) ([]Author, error) {
  rows, err := q.db.QueryContext(ctx, listAuthors, pq.Array(ids))
  if err != nil {
    return nil, err
  }
  defer rows.Close()
  var items []Author
  for rows.Next() {
    var i Author
    if err := rows.Scan(&i.ID, &i.Bio, &i.BirthYear); err != nil {
      return nil, err
    }
    items = append(items, i)
  }
  if err := rows.Close(); err != nil {
    return nil, err
  }
  if err := rows.Err(); err != nil {
    return nil, err
  }
  return items, nil
}
```

CHAPTER

FOUR

COUNTING ROWS

```sql
CREATE TABLE authors (
  id       SERIAL PRIMARY KEY,
  hometown text   NOT NULL
);

-- name: CountAuthors :one
SELECT count(*) FROM authors;

-- name: CountAuthorsByTown :many
SELECT hometown, count(*) FROM authors
GROUP BY 1
ORDER BY 1;
```

```go
package db

import (
	"context"
	"database/sql"
)

type DBTX interface {
	QueryContext(context.Context, string, ...interface{}) (*sql.Rows, error)
	QueryRowContext(context.Context, string, ...interface{}) *sql.Row
}

func New(db DBTX) *Queries {
	return &Queries{db: db}
}

type Queries struct {
	db DBTX
}

const countAuthors = `-- name: CountAuthors :one
SELECT count(*) FROM authors
`

func (q *Queries) CountAuthors(ctx context.Context) (int, error) {
	row := q.db.QueryRowContext(ctx, countAuthors)
	var i int
	err := row.Scan(&i)
	return i, err
}
```

(continues on next page)

(continued from previous page)

```
const countAuthorsByTown = `-- name: CountAuthorsByTown :many
SELECT hometown, count(*) FROM authors
GROUP BY 1
ORDER BY 1
`

type CountAuthorsByTownRow struct {
	Hometown  string
	Count     int
}

func (q *Queries) CountAuthorsByTown(ctx context.Context) ([]CountAuthorsByTownRow,␣
↪error) {
	rows, err := q.db.QueryContext(ctx, countAuthorsByTown)
	if err != nil {
		return nil, err
	}
	defer rows.Close()
	items := []CountAuthorsByTownRow{}
	for rows.Next() {
		var i CountAuthorsByTownRow
		if err := rows.Scan(&i.Hometown, &i.Count); err != nil {
			return nil, err
		}
		items = append(items, i)
	}
	if err := rows.Close(); err != nil {
		return nil, err
	}
	if err := rows.Err(); err != nil {
		return nil, err
	}
	return items, nil
}
```

CHAPTER

FIVE

INSERTING ROWS

```
CREATE TABLE authors (
  id          SERIAL PRIMARY KEY,
  bio         text   NOT NULL
);

-- name: CreateAuthor :exec
INSERT INTO authors (bio) VALUES ($1);
```

```
package db

import (
        "context"
        "database/sql"
)

type DBTX interface {
        ExecContext(context.Context, string, ...interface{}) error
}

func New(db DBTX) *Queries {
        return &Queries{db: db}
}

type Queries struct {
        db DBTX
}

const createAuthor = `-- name: CreateAuthor :exec
INSERT INTO authors (bio) VALUES ($1)
`

func (q *Queries) CreateAuthor(ctx context.Context, bio string) error {
        _, err := q.db.ExecContext(ctx, createAuthor, bio)
        return err
}
```

5.1 Returning columns from inserted rows

sqlc has full support for the `RETURNING` statement.

```
CREATE TABLE authors (
  id          SERIAL PRIMARY KEY,
  bio         text   NOT NULL
);

-- name: Delete :exec
DELETE FROM authors WHERE id = $1;

-- name: DeleteAffected :execrows
DELETE FROM authors WHERE id = $1;

-- name: DeleteID :one
DELETE FROM authors WHERE id = $1
RETURNING id;

-- name: DeleteAuthor :one
DELETE FROM authors WHERE id = $1
RETURNING *;
```

```
package db

import (
	"context"
	"database/sql"
)

type Author struct {
	ID  int
	Bio string
}

type DBTX interface {
	ExecContext(context.Context, string, ...interface{}) error
	QueryRowContext(context.Context, string, ...interface{}) error
}

func New(db DBTX) *Queries {
	return &Queries{db: db}
}

type Queries struct {
	db DBTX
}

const delete = `-- name: Delete :exec
DELETE FROM authors WHERE id = $1
`

func (q *Queries) Delete(ctx context.Context, id int) error {
	_, err := q.db.ExecContext(ctx, delete, id)
	return err
}
```

(continues on next page)

(continued from previous page)

```
const deleteAffected = `-- name: DeleteAffected :exec
DELETE FROM authors WHERE id = $1
`

func (q *Queries) DeleteAffected(ctx context.Context, id int) (int64, error) {
        result, err := q.db.ExecContext(ctx, deleteAffected, id)
        if err != nil {
                return 0, err
        }
        return result.RowsAffected()
}

const deleteID = `-- name: DeleteID :one
DELETE FROM authors WHERE id = $1
RETURNING id
`

func (q *Queries) DeleteID(ctx context.Context, id int) (int, error) {
        row := q.db.QueryRowContext(ctx, deleteID, id)
        var i int
        err := row.Scan(&i)
        return i, err
}

const deleteAuhtor = `-- name: DeleteAuthor :one
DELETE FROM authors WHERE id = $1
RETURNING id, bio
`

func (q *Queries) DeleteAuthor(ctx context.Context, id int) (Author, error) {
        row := q.db.QueryRowContext(ctx, deleteAuhtor, id)
        var i Author
        err := row.Scan(&i.ID, &i.Bio)
        return i, err
}
```

CHAPTER

SIX

UPDATING ROWS

```
CREATE TABLE authors (
  id          SERIAL PRIMARY KEY,
  bio         text   NOT NULL
);

-- name: UpdateAuthor :exec
UPDATE authors SET bio = $2
WHERE id = $1;
```

```
package db

import (
        "context"
        "database/sql"
)

type DBTX interface {
        ExecContext(context.Context, string, ...interface{}) error
}

func New(db DBTX) *Queries {
        return &Queries{db: db}
}

type Queries struct {
        db DBTX
}

const updateAuthor = `-- name: UpdateAuthor :exec
UPDATE authors SET bio = $2
WHERE id = $1
`

func (q *Queries) UpdateAuthor(ctx context.Context, id int, bio string) error {
        _, err := q.db.ExecContext(ctx, updateAuthor, id, bio)
        return err
}
```

CHAPTER

SEVEN

DELETING ROWS

```
CREATE TABLE authors (
  id          SERIAL PRIMARY KEY,
  bio         text   NOT NULL
);

-- name: DeleteAuthor :exec
DELETE FROM authors WHERE id = $1;
```

```
package db

import (
	"context"
	"database/sql"
)

type DBTX interface {
	ExecContext(context.Context, string, ...interface{}) error
}

func New(db DBTX) *Queries {
	return &Queries{db: db}
}

type Queries struct {
	db DBTX
}

const deleteAuthor = `-- name: DeleteAuthor :exec
DELETE FROM authors WHERE id = $1
`

func (q *Queries) DeleteAuthor(ctx context.Context, id int) error {
	_, err := q.db.ExecContext(ctx, deleteAuthor, id)
	return err
}
```

CHAPTER

EIGHT

PREPARING QUERIES

```
CREATE TABLE records (
  id SERIAL PRIMARY KEY
);

-- name: GetRecord :one
SELECT * FROM records
WHERE id = $1;
```

sqlc has an option to use perpared queries. These prepared queries also work with transactions.

```
package db

import (
        "context"
        "database/sql"
)

type Record struct {
        ID int
}

type DBTX interface {
        PrepareContext(context.Context, string) (*sql.Stmt, error)
        QueryRowContext(context.Context, string, ...interface{}) *sql.Row
}

func New(db DBTX) *Queries {
        return &Queries{db: db}
}

func Prepare(ctx context.Context, db DBTX) (*Queries, error) {
        q := Queries{db: db}
        var err error
        if q.getRecordStmt, err = db.PrepareContext(ctx, getRecord); err != nil {
                return nil, err
        }
        return &q, nil
}

func (q *Queries) queryRow(ctx context.Context, stmt *sql.Stmt, query string, args ...
↪interface{}) (*sql.Row) {
        switch {
        case stmt != nil && q.tx != nil:
                return q.tx.StmtContext(ctx, stmt).QueryRowContext(ctx, args...)
```

(continues on next page)

(continued from previous page)

```
	case stmt != nil:
		return stmt.QueryRowContext(ctx, args...)
	default:
		return q.db.QueryRowContext(ctx, query, args...)
	}
}

type Queries struct {
	db            DBTX
	tx            *sql.Tx
	getRecordStmt *sql.Stmt
}

func (q *Queries) WithTx(tx *sql.Tx) *Queries {
	return &Queries{
		db:            tx,
		tx:            tx,
		getRecordStmt: q.getRecordStmt,
	}
}

const getRecord = `-- name: GetRecord :one
SELECT id FROM records
WHERE id = $1
`

func (q *Queries) GetRecord(ctx context.Context, id int) (Record, error) {
	row := q.queryRow(ctx, q.getRecordStmt, getRecord, id)
	var i Record
	err := row.Scan(&i.ID)
	return i, err
}
```

CHAPTER

NINE

USING TRANSACTIONS

```
CREATE TABLE records (
  id SERIAL PRIMARY KEY
);

-- name: GetRecord :one
SELECT * FROM records
WHERE id = $1;
```

The `WithTx` method allows a `Queries` instance to be associated with a transaction.

```
package db

import (
        "context"
        "database/sql"
)

type Record struct {
        ID int
}

type DBTX interface {
        QueryRowContext(context.Context, string, ...interface{}) *sql.Row
}

func New(db DBTX) *Queries {
        return &Queries{db: db}
}

type Queries struct {
        db DBTX
}

func (*Queries) WithTx(tx *sql.Tx) *Queries {
        return &Queries{db: tx}
}

const getRecord = `-- name: GetRecord :one
SELECT id FROM records
WHERE id = $1
`

func (q *Queries) GetRecord(ctx context.Context, id int) (Record, error) {
        row := q.db.QueryRowContext(ctx, getRecord, id)
```

(continues on next page)

(continued from previous page)

```
        var i Record
        err := row.Scan(&i.ID)
        return i, err
}
```

CHAPTER

TEN

NAMING PARAMETERS

sqlc tried to generate good names for positional parameters, but sometimes it lacks enough context. The following SQL generates parameters with less than ideal names:

```
-- name: UpsertAuthorName :one
UPDATE author
SET
  name = CASE WHEN $1::bool
    THEN $2::text
    ELSE name
    END
RETURNING *;
```

```
type UpdateAuthorNameParams struct {
  Column1        bool          `json:""`
  Column2_2      string        `json:"_2"`
}
```

In these cases, named parameters give you the control over field names on the Params struct.

```
-- name: UpsertAuthorName :one
UPDATE author
SET
  name = CASE WHEN sqlc.arg(set_name)::bool
    THEN sqlc.arg(name)::text
    ELSE name
    END
RETURNING *;
```

```
type UpdateAuthorNameParams struct {
  SetName        bool          `json:"set_name"`
  Name           string        `json:"name"`
}
```

If the `sqlc.arg()` syntax is too verbose for your taste, you can use the `@` operator as a shortcut.

```
-- name: UpsertAuthorName :one
UPDATE author
SET
  name = CASE WHEN @set_name::bool
    THEN @name::text
    ELSE name
    END
RETURNING *;
```

CHAPTER

ELEVEN

MODIFYING THE DATABASE SCHEMA

sqlc understands ALTER TABLE statements when parsing SQL.

```
CREATE TABLE authors (
  id          SERIAL PRIMARY KEY,
  birth_year  int    NOT NULL
);

ALTER TABLE authors ADD COLUMN bio text NOT NULL;
ALTER TABLE authors DROP COLUMN birth_year;
ALTER TABLE authors RENAME TO writers;
```

```
package db

type Writer struct {
        ID  int
        Bio string
}
```

11.1 Handling SQL migrations

sqlc will ignore rollback statements when parsing migration SQL files. The following tools are current supported:

- dbmate
- golang-migrate
- goose
- sql-migrate
- tern

11.1.1 goose

```
-- +goose Up
CREATE TABLE post (
    id    int NOT NULL,
    title text,
    body  text,
    PRIMARY KEY(id)
);
```

(continues on next page)

(continued from previous page)

```

-- +goose Down
DROP TABLE post;
```

```
package db

type Post struct {
        ID    int
        Title sql.NullString
        Body  sql.NullString
}
```

11.1.2 sql-migrate

```
-- +migrate Up
-- SQL in section 'Up' is executed when this migration is applied
CREATE TABLE people (id int);


-- +migrate Down
-- SQL section 'Down' is executed when this migration is rolled back
DROP TABLE people;
```

```
package db

type People struct {
        ID    int32
}
```

11.1.3 tern

```
CREATE TABLE comment (id int NOT NULL, text text NOT NULL);
---- create above / drop below ----
DROP TABLE comment;
```

```
package db

type Comment struct {
        ID     int32
        Text   string
}
```

11.1.4 golang-migrate

In `20060102.up.sql`:

```sql
CREATE TABLE post (
    id    int NOT NULL,
    title text,
    body  text,
    PRIMARY KEY(id)
);
```

In `20060102.down.sql`:

```sql
DROP TABLE post;
```

```go
package db

type Post struct {
        ID    int
        Title sql.NullString
        Body  sql.NullString
}
```

11.1.5 dbmate

```sql
-- migrate:up
CREATE TABLE foo (bar INT NOT NULL);

-- migrate:down
DROP TABLE foo;
```

```go
package db

type Foo struct {
        Bar int32
}
```

CHAPTER

TWELVE

CONFIGURING GENERATED STRUCTS

12.1 Naming scheme

Structs generated from tables will attempt to use the singular form of a table name if the table name is pluralized.

```
CREATE TABLE authors (
  id   SERIAL PRIMARY KEY,
  name text   NOT NULL
);
```

```
package db

// Struct names use the singular form of table names
type Author struct {
	ID   int
	Name string
}
```

12.2 JSON tags

```
CREATE TABLE authors (
  id         SERIAL    PRIMARY KEY,
  created_at timestamp NOT NULL
);
```

sqlc can generate structs with JSON tags. The JSON name for a field matches the column name in the database.

```
package db

import (
	"time"
)

type Author struct {
	ID        int       `json:"id"`
	CreatedAt time.Time `json:"created_at"`
}
```

CHAPTER

THIRTEEN

CLI

```
Usage:
  sqlc [command]

Available Commands:
  compile     Statically check SQL for syntax and type errors
  generate    Generate Go code from SQL
  help        Help about any command
  init        Create an empty sqlc.yaml settings file
  version     Print the sqlc version number

Flags:
  -h, --help   help for sqlc

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

CHAPTER FOURTEEN

CONFIGURATION FILE (VERSION 1)

The `sqlc` tool is configured via a `sqlc.yaml` or `sqlc.json` file. This file must be in the directory where the `sqlc` command is run.

```yaml
version: "1"
packages:
  - name: "db"
    path: "internal/db"
    queries: "./sql/query/"
    schema: "./sql/schema/"
    engine: "postgresql"
    emit_json_tags: true
    emit_prepared_queries: true
    emit_interface: false
    emit_exact_table_names: false
    emit_empty_slices: false
```

Each package document has the following keys:

- `name`:
  - The package name to use for the generated code. Defaults to `path` basename
- `path`:
  - Output directory for generated code
- `queries`:
  - Directory of SQL queries or path to single SQL file; or a list of paths
- `schema`:
  - Directory of SQL migrations or path to single SQL file; or a list of paths
- `engine`:
  - Either `postgresql` or `mysql`. Defaults to `postgresql`. MySQL support is experimental
- `emit_json_tags`:
  - If true, add JSON tags to generated structs. Defaults to `false`.
- `emit_db_tags`:
  - If true, add DB tags to generated structs. Defaults to `false`.
- `emit_prepared_queries`:
  - If true, include support for prepared queries. Defaults to `false`.
- `emit_interface`:

    - If true, output a `Querier` interface in the generated package. Defaults to `false`.

- `emit_exact_table_names`:
    - If true, struct names will mirror table names. Otherwise, sqlc attempts to singularize plural table names. Defaults to `false`.
- `emit_empty_slices`:
    - If true, slices returned by `:many` queries will be empty instead of `nil`. Defaults to `false`.

14.1 Type Overrides

The default mapping of PostgreSQL types to Go types only uses packages outside the standard library when it must.

For example, the `uuid` PostgreSQL type is mapped to `github.com/google/uuid`. If a different Go package for UUIDs is required, specify the package in the `overrides` array. In this case, I'm going to use the `github.com/gofrs/uuid` instead.

```yaml
version: "1"
packages: [...]
overrides:
  - go_type: "github.com/gofrs/uuid.UUID"
    db_type: "uuid"
```

Each override document has the following keys:

- `db_type`:
    - The PostgreSQL type to override. Find the full list of supported types in postgresql_type.go.
- `go_type`:
    - A fully qualified name to a Go type to use in the generated code.
- `nullable`:
    - If true, use this type when a column is nullable. Defaults to `false`.

14.2 Per-Column Type Overrides

Sometimes you would like to override the Go type used in model or query generation for a specific field of a table and not on a type basis as described in the previous section.

This may be configured by specifying the `column` property in the override definition. `column` should be of the form `table.column` buy you may be even more specify by specifying `schema.table.column` or `catalog.schema.table.column`.

```yaml
version: "1"
packages: [...]
overrides:
  - column: "authors.id"
    go_type: "github.com/segmentio/ksuid.KSUID"
```

14.3 Package Level Overrides

Overrides can be configured globally, as demonstrated in the previous sections, or they can be configured on a per-package which scopes the override behavior to just a single package:

```
version: "1"
packages:
  - overrides: [...]
```

14.4 Renaming Struct Fields

Struct field names are generated from column names using a simple algorithm: split the column name on underscores and capitalize the first letter of each part.

```
account     -> Account
spotify_url -> SpotifyUrl
app_id      -> AppID
```

If you're not happy with a field's generated name, use the `rename` dictionary to pick a new name. The keys are column names and the values are the struct field name to use.

```
version: "1"
packages: [...]
rename:
  spotify_url: "SpotifyURL"
```

CHAPTER

FIFTEEN

DATATYPES

15.1 Arrays

PostgreSQL arrays are materialized as Go slices. Currently, only one-dimensional arrays are supported.

```
CREATE TABLE places (
  name text   not null,
  tags text[]
);
```

```
package db

type Place struct {
        Name string
        Tags []string
}
```

15.2 Dates and Time

All PostgreSQL time and date types are returned as `time.Time` structs. For null time or date values, the `NullTime` type from `database/sql` is used.

```
CREATE TABLE authors (
  id         SERIAL    PRIMARY KEY,
  created_at timestamp NOT NULL DEFAULT NOW(),
  updated_at timestamp
);
```

```
package db

import (
        "time"
        "database/sql"
)

type Author struct {
        ID        int
        CreatedAt time.Time
        UpdatedAt sql.NullTime
}
```

15.3 Enums

PostgreSQL enums are mapped to an aliased string type.

```sql
CREATE TYPE status AS ENUM (
  'open',
  'closed'
);

CREATE TABLE stores (
  name   text    PRIMARY KEY,
  status status  NOT NULL
);
```

```go
package db

type Status string

const (
        StatusOpen   Status = "open"
        StatusClosed Status = "closed"
)

type Store struct {
        Name   string
        Status Status
}
```

15.4 Null

For structs, null values are represented using the appropriate type from the `database/sql` package.

```sql
CREATE TABLE authors (
  id   SERIAL PRIMARY KEY,
  name text   NOT NULL,
  bio  text
);
```

```go
package db

import (
        "database/sql"
)

type Author struct {
        ID   int
        Name string
        Bio  sql.NullString
}
```

15.5 UUIDs

The Go standard library does not come with a `uuid` package. For UUID support, sqlc uses the excellent `github.com/google/uuid` package.

```
CREATE TABLE records (
  id   uuid PRIMARY KEY
);
```

```
package db

import (
	"github.com/google/uuid"
)

type Author struct {
	ID   uuid.UUID
}
```

CHAPTER SIXTEEN

QUERY ANNOTATIONS

sqlc requires each query to have a small comment indicating the name and command. The format of this comment is as follows:

```
-- name: <name> <command>
```

16.1 :exec

The generated method will return the error from ExecContext.

```
-- name: DeleteAuthor :exec
DELETE FROM authors
WHERE id = $1;
```

```
func (q *Queries) DeleteAuthor(ctx context.Context, id int64) error {
  _, err := q.db.ExecContext(ctx, deleteAuthor, id)
  return err
}
```

16.2 :execresult

The generated method will return the sql.Result returned by ExecContext.

```
-- name: DeleteAllAuthors :execresult
DELETE FROM authors;
```

```
func (q *Queries) DeleteAllAuthors(ctx context.Context) (sql.Result, error) {
  return q.db.ExecContext(ctx, deleteAllAuthors)
}
```

16.3 :execrows

The generated method will return the number of affected rows from the result returned by ExecContext.

```sql
-- name: DeleteAllAuthors :execrows
DELETE FROM authors;
```

```go
func (q *Queries) DeleteAllAuthors(ctx context.Context) (int64, error) {
  _, err := q.db.ExecContext(ctx, deleteAllAuthors)
  // ...
}
```

16.4 :many

The generated method will return a slice of records via QueryContext.

```sql
-- name: ListAuthors :many
SELECT * FROM authors
ORDER BY name;
```

```go
func (q *Queries) ListAuthors(ctx context.Context) ([]Author, error) {
  rows, err := q.db.QueryContext(ctx, listAuthors)
  // ...
}
```

16.5 :one

The generated method will return a single record via QueryRowContext.

```
-- name: GetAuthor :one
SELECT * FROM authors
WHERE id = $1 LIMIT 1;
```

```go
func (q *Queries) GetAuthor(ctx context.Context, id int64) (Author, error) {
  row := q.db.QueryRowContext(ctx, getAuthor, id)
  // ...
}
```

CHAPTER

SEVENTEEN

DEVELOPING SQLC

17.1 Building

For local development, install `sqlc` under an alias. We suggest `sqlc-dev`.

```
go build -o ~/go/bin/sqlc-dev ./cmd/sqlc
```

17.2 Running Tests

```
go test ./...
```

To run the tests in the examples folder, use the `examples` tag.

```
go test --tags=examples ./...
```

These tests require locally-running database instances. Run these databases using Docker Compose.

```
docker-compose up -d
```

The tests use the following environment variables to connect to the database

17.2.1 For PostgreSQL

```
Variable     Default Value
-------------------------
PG_HOST      127.0.0.1
PG_PORT      5432
PG_USER      postgres
PG_PASSWORD  mysecretpassword
PG_DATABASE  dinotest
```

17.2.2 For MySQL

```
Variable     Default Value
-------------------------
MYSQL_HOST      127.0.0.1
MYSQL_PORT      3306
MYSQL_USER      root
MYSQL_ROOT_PASSWORD  mysecretpassword
MYSQL_DATABASE  dinotest
```

17.3 Regenerate expected test output

If you need to update a large number of expected test output in the `internal/endtoend/testdata` directory, run the `regenerate.sh` script.

```
make regen
```

Note that this uses the `sqlc-dev` binary, not `sqlc` so make sure you have an up to date `sqlc-dev` binary.

CHAPTER

EIGHTEEN

PRIVACY AND DATA COLLECTION

These days, it feels like every piece of software is tracking you. From your browser, to your phone, to your terminal, programs collect as much data about you as possible and send it off to the cloud for analysis.

We believe the best way to keep data safe is to never collect it in the first place.

18.1 Our Privacy Pledge

The `sqlc` program does not collect any information. It does not send crash reports to a third-party. It does not gather anonymous aggregate user behaviour analytics.

No analytics. No finger-printing. No tracking.

Not now and not in the future.

18.2 Distribution Channels

We distribute sqlc using popular package managers such as Homebrew and Snapcraft. These package managers and their associated command-line tools do collect usage metrics.

We use these services to make it easy to for users to install sqlc. There will always be an option to download sqlc from a stable URL.