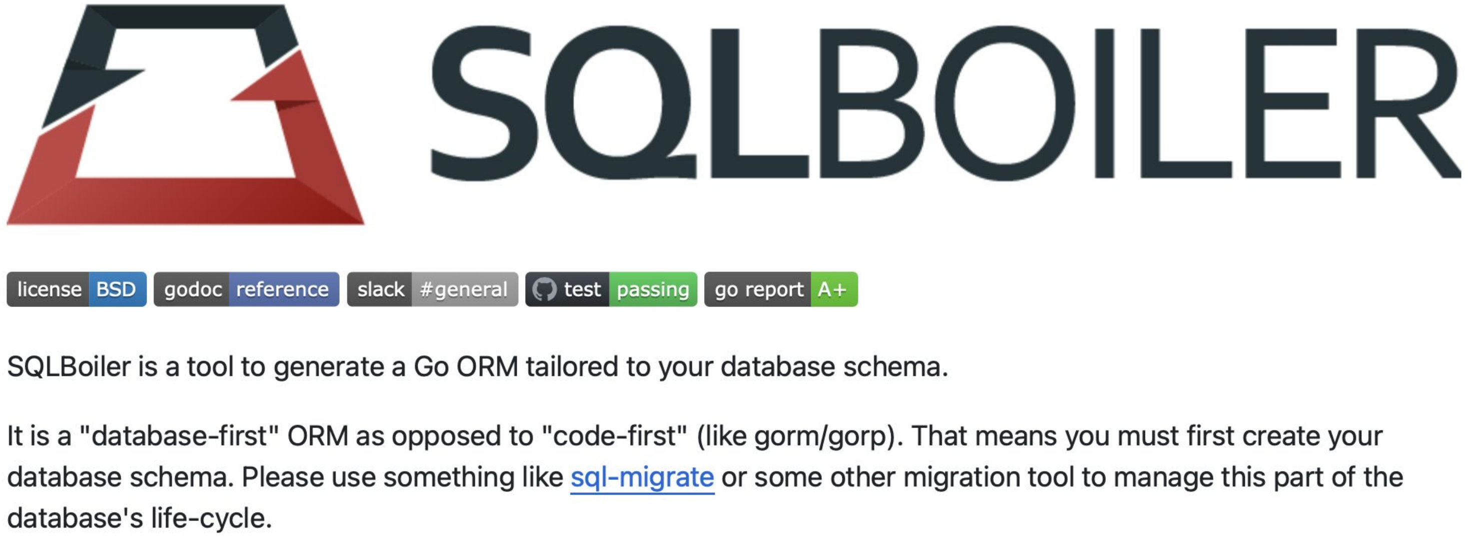
sqlboiler

The Good, The Bad and The Ugly

jakubpieta

ORM

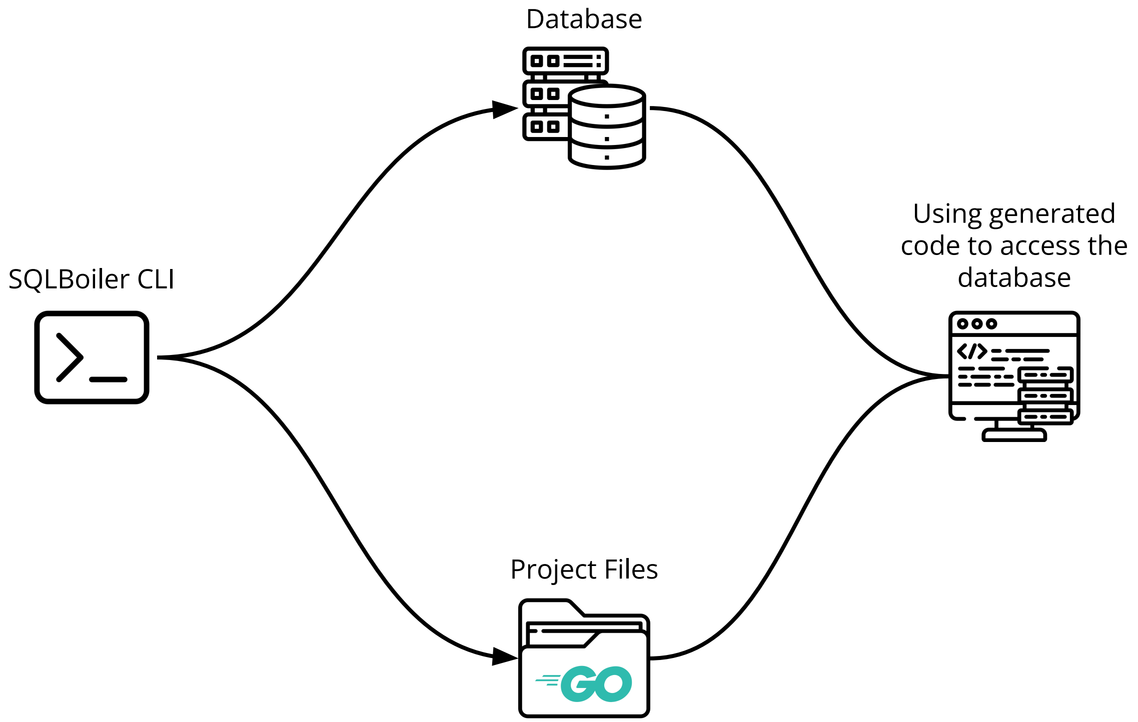
(Object-Relational Mapping)



SQLBoiler is a tool to generate a Go ORM tailored to your database schema.

It is a "database-first" ORM as opposed to "code-first" (like gorm/gorp). That means you must first create your database schema. Please use something like sql-migrate or some other migration tool to manage this part of the database's life-cycle.

https://github.com/volatiletech/sqlboiler


Database
SQLBoiler CLI
Using generated code to access the database
Project Files
GO

DB Config

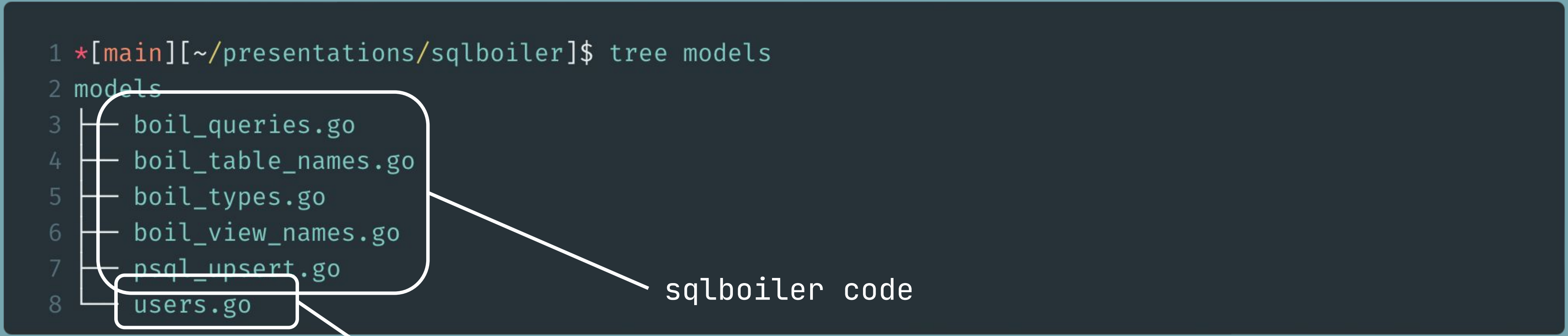
sqlboiler.toml

```toml
output   = "models"
no-tests = true
wipe     = true

[psql]
    dbname = "sqlboiler"
    host   = "localhost"
    port   = 30000
    user   = "dbadmin"
    pass   = "admin123"
    sslmode = "disable"
```

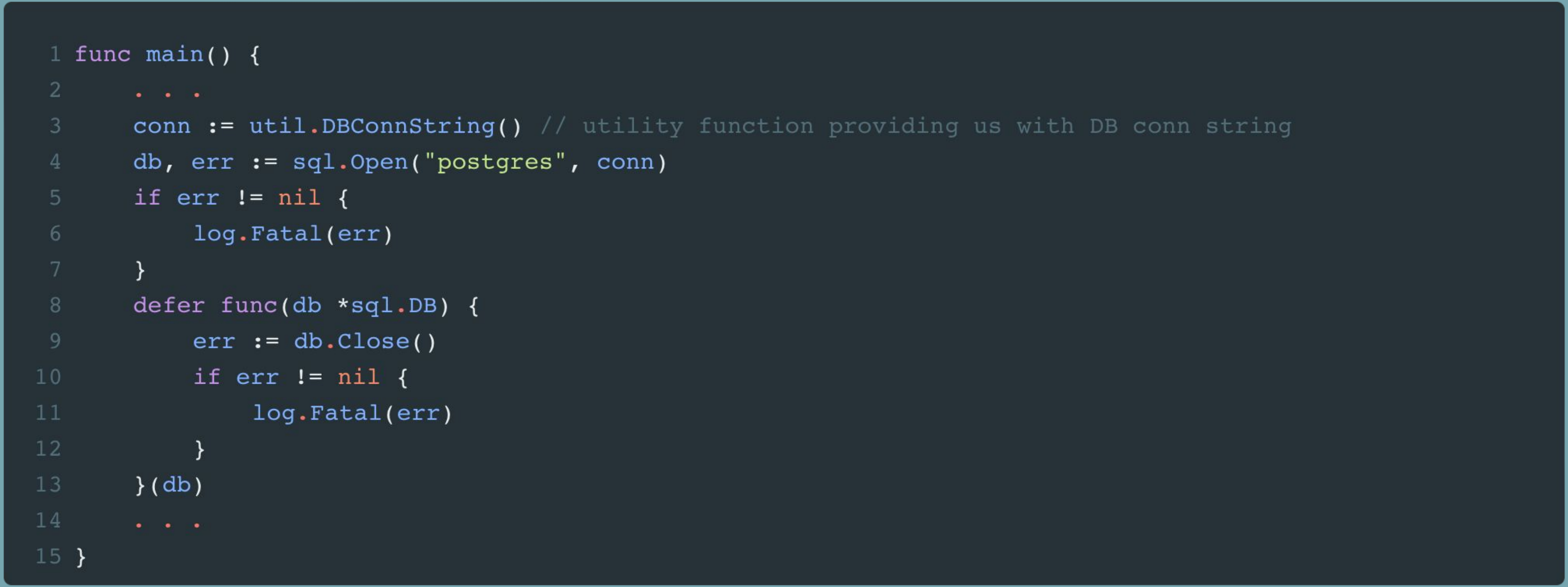
migration.sql

```sql
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";

CREATE TABLE users (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    username VARCHAR(255) UNIQUE NOT NULL,
    email VARCHAR(320) UNIQUE NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

```
# Go 1.16 and above:
$ go install github.com/volatiletech/sqlboiler/v4@latest
$ go install github.com/volatiletech/sqlboiler/v4/drivers/sqlboiler-psql@latest

$ sqlboiler psql
```



```
*[main][~/presentations/sqlboiler]$ tree models
models
├── boil_queries.go
├── boil_table_names.go
├── boil_types.go
├── boil_view_names.go
├── psql_upsert.go
└── users.go
```



```go
func main() {
    . . .
    conn := util.DBConnString() // utility function providing us with DB conn string
    db, err := sql.Open("postgres", conn)
    if err != nil {
        log.Fatal(err)
    }
    defer func(db *sql.DB) {
        err := db.Close()
        if err != nil {
            log.Fatal(err)
        }
    }(db)
    . . .
}
```

The Good

schema oriented code

```
*[main][~/presentations/sqlboiler]$ tree models
models
├── boil_queries.go
├── boil_table_names.go
├── boil_types.go
├── boil_view_names.go
├── psql_upsert.go
└── users.go
```

our own schema specific code

abstraction of database complexity

users.go

```go
// User is an object representing the database table.
type User struct {
    ID        string    `boil:"id" json:"id" ...`
    Username  string    `boil:"username" json:"username" ...`
    Email     string    `boil:"email" json:"email" ...`
    CreatedAt null.Time `boil:"created_at" json:"created_at,omitempty" ...`
    UpdatedAt null.Time `boil:"updated_at" json:"updated_at,omitempty" ...`
}
```

raw SQL approach

```go
type User struct {
    ID       int
    Username string
    Email    string
}

func getUserByIDRawSQL(db *sql.DB, userID string) (*User, error) {
    query := "SELECT id, username, email FROM users WHERE id = $1"
    row := db.QueryRowContext(context.Background(), query, userID)

    var user User
    err := row.Scan(&user.ID, &user.Username, &user.Email)
    if err ≠ nil {
        return nil, err
    }
    return &user, nil
}
```

raw SQL approach

```go
func main() {
    . . .
    userID := "some-uuid"
    user, err := getUserByIDRawSQL(db, userID)
    if err ≠ nil {
        log.Fatal(err)
    }
    . . .
}
```

sqlboiler

```go
import (
    . . .
    "your/package/models"
    . . .
)

func main() {
    userID := "some-uuid"
    user, err := models.Users(
        models.UserWhere.ID.EQ(userID),
    ).One(context.Background(), boil.GetDB())
    if err ≠ nil {
        log.Fatal(err)
    }
}
```

CRUD and other operations

```go
err := user.Insert(ctx, db, boil.Infer())

user.Username = "new_example_username"
updatedCount, err := user.Update(ctx, db, boil.Infer())

user, err = models.Users(
    models.UserWhere.Username.EQ("new_example_username"),
).One(ctx, db) // similar to Users.All()

rowsDeleted, err := user.Delete(ctx, db) // similar to Users.DeleteAll()
```

utility functions

```go
func main() {
  . . .
  complexQueryUsers, err := models.Users(
    models.UserWhere.Email.EQ("example@example.com"),
    models.UserWhere.CreatedAt.GT(null.TimeFrom(time.Now())),
    models.UserWhere.Username.LIKE("example%"),
    models.UserWhere.UpdatedAt.LT(null.TimeFrom(time.Now())),
    models.UserWhere.ID.IN([]string{"550e8400-e29b-41d4-a716-446655440000"}),
  ).All(ctx, db)
  if err != nil {
    return
  }
  . . .
}
```

relations

```sql
CREATE TABLE posts (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    title VARCHAR(255) NOT NULL,
    content TEXT,
    user_id UUID REFERENCES users(id),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

```go
// user.go
type User struct {
  ID        string    `boil:"id" ...`
  . . .

  R *userR `boil:"-" json:"-" toml:"-" yaml:"-"`
  L userL  `boil:"-" json:"-" toml:"-" yaml:"-"`
}

// post.go
type Post struct {
    ID        string      `boil:"id" . . .`
    Title     string      `boil:"title". . .`
    Content   null.String `boil:"content". . .`
    UserID    null.String `boil:"user_id". . .`
    CreatedAt null.Time   `boil:"created_at". . .`
    UpdatedAt null.Time   `boil:"updated_at". . .`

    R *postR `boil:"-" json:"-" toml:"-" yaml:"-"`
    L postL  `boil:"-" json:"-" toml:"-" yaml:"-"`
}
```

relations

```go
func main() {
    ...
    user, err := models.Users(
        qm.Where(models.UserColumns.Username+"=?", "john_doe"),
        qm.Load(qm.Rels(models.UserRels.Posts)),
    ).One(ctx, db)
    if err != nil {
        log.Fatal(err)
    }
    ...
}
```

relations

```go
user, err := models.Users(
    qm.Where(models.UserColumns.Username+"=?", "john_doe"),
    qm.Load(qm.Rels(models.UserRels.Posts)),
  ).One(ctx, db)
if err ≠ nil {
  log.Fatal(err)
}

for _, post := range user.R.Posts {
  // do something
}
```

lazy-loading relations

```
SELECT "users".* FROM "users" WHERE (username=$1) LIMIT 1;
[john_doe]
SELECT * FROM "posts" WHERE ("posts"."user_id" IN ($1));
[4f82de9d-00c4-4b4d-a66b-73dc97d94cfa]
```

(it’s not perfect, but in most cases it’s enough) less boilerplate written

(more boilerplate generated)

easier than SQL learning curve

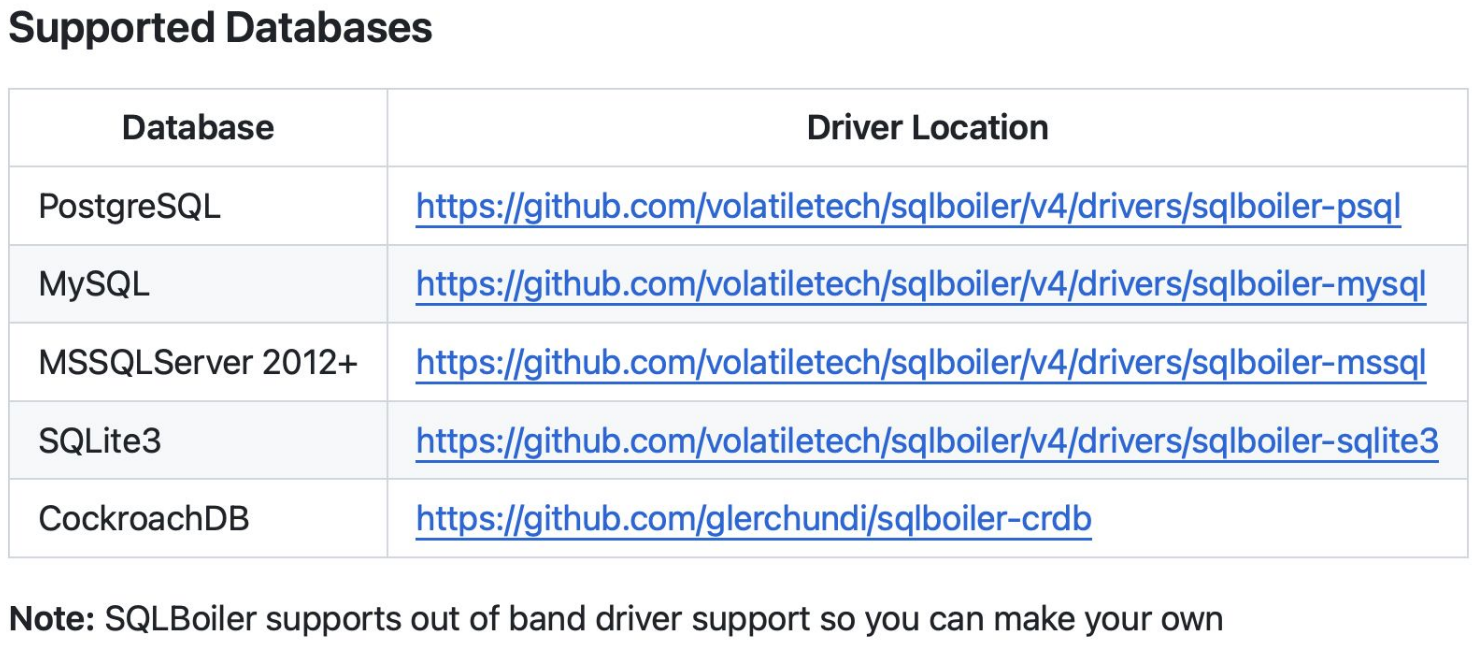
(at the beginning)

shorter development time

(when you don't have complex aggregates and relations)

easier prototyping and delivering MVP

portability



Supported Databases

Database	Driver Location
PostgreSQL	https://github.com/volatiletech/sqlboiler/v4/drivers/sqlboiler-psql
MySQL	https://github.com/volatiletech/sqlboiler/v4/drivers/sqlboiler-mysql
MSSQLServer 2012+	https://github.com/volatiletech/sqlboiler/v4/drivers/sqlboiler-mssql
SQLite3	https://github.com/volatiletech/sqlboiler/v4/drivers/sqlboiler-sqlite3
CockroachDB	https://github.com/glerchundi/sqlboiler-crdb

Note: SQLBoiler supports out of band driver support so you can make your own

The Bad

more relations

```sql
CREATE TABLE comments (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    content TEXT NOT NULL,
    user_id UUID REFERENCES users(id),
    post_id UUID REFERENCES posts(id),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE ratings (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    value INTEGER NOT NULL CHECK (value >= 1 AND value <= 5),
    user_id UUID REFERENCES users(id),
    post_id UUID REFERENCES posts(id),
    comment_id UUID REFERENCES comments(id),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

lazy-loading when you need *everything*

```go
user, err := models.Users(
  models.UserWhere.Username.EQ("john_doe"),
  qm.Load(qm.Rels(
    models.UserRels.Posts,
    models.PostRels.Ratings,
    models.RatingRels.Comment,
  )),
).One(ctx, db)
if err ≠ nil {
  log.Fatal(err)
}
```

lazy-loading when you need *everything*

```
SELECT "users".* FROM "users" WHERE ("users"."username" = $1) LIMIT 1;
[john_doe]
SELECT * FROM "posts" WHERE ("posts"."user_id" IN ($1));
[d1e77214-f181-4196-b857-818aff6d1ee3]
SELECT * FROM "ratings" WHERE ("ratings"."post_id" IN ($1));
[a6a97426-0320-40de-a099-f93d116cc30d]
SELECT * FROM "comments" WHERE ("comments"."post_id" IN ($1));
[a6a97426-0320-40de-a099-f93d116cc30d]
```

(eager loading here would look ugly with sqlboiler)

eager-loading

```go
post, err := models.Posts(
    qm.InnerJoin("users on users.id = posts.user_id"),
    models.UserWhere.Username.EQ("john_doe"),
  ).One(ctx, db)
if err != nil {
  log.Fatal(err)
}
```

```sql
SELECT "posts".* FROM "posts"
INNER JOIN users on users.id = posts.user_id
WHERE ("users"."username" = $1) LIMIT 1;
[john_doe]
```

(it doesn't work out of the box)

eager-loading

```go
type PostWithUser struct {
  models.Post `boil:"posts,bind"`
  models.User `boil:"users,bind"`
}

var userWithPost PostWithUser
err := models.NewQuery(
    qm.Select("users.*", "posts.*"),
    qm.From(models.TableNames.Posts),
    qm.InnerJoin("users on users.id = posts.user_id"),
    models.UserWhere.Username.EQ("john_doe"),
).Bind(ctx, db, &userWithPost)
if err ≠ nil {
  log.Fatal(err)
}
```

sqlboiler bindings don’t support **null** values in left/right/full joins

and The Ugly

how it actually looks like

(time to dive into IDE)

The Vietnam of Computer Science

By Ted Neward June 26, 2006

https://www.odbms.org/wp-content/uploads/2013/11/031.01-Neward-The-Vietnam-of-Computer-Science-June-2006.pdf

objects and relations

(fundamentally different, yet we need to combine them)

object oriented system aims to provide:

1. identity management
2. state management
3. behaviors
4. encapsulation
5. polymorphism

relational database aim to:

1. normalize data
2. represent facts
3. **ACID** compliance
4. perform SQL operations based on set theory
5. achieve normalization through proper design

because of the differences
you eventually stop leveraging
your database capabilities / oop principles

(and that's pretty sad)

jakubpieta

Thank you!

sources:

- SQLBoiler
- Third normal form
- Object-oriented programming
- The Vietnam of Computer Science
- Exiting the Vietnam of Programming: Our Journey in Dropping the ORM (in Golang)

you can find the code examples on my GitHub

jakubpieta
SWE@nobl9
Poznań
j.pieta96@gmail.com