Nikolay Kuznetsov

Senior Software Engineer

Beyond sqlc: teaching AI to generate repositories and integration tests

About me

- Senior Software Engineer
- Zalando Helsinki 🇫🇮
- C → Java → Kotlin → Go
- Author of *pgx-outbox* and *sqlc++* projects

Goals

- Create repositories on top of *sqlc*-generated queries
- Keep *sqlc*'s strengths: type safety & compile-time checks
- Decouple business logic from *sqlc*-generated records
- Automate repository creation with reference-based AI generation

Repository

- Accepts and returns domain models
- Hides details of SQL, schema, and query libraries
- Maps between DB records and domain models
- Orchestrates transactions across multiple operations

- Cart repository interface

```
type CartRepository interface {
    // context.Context type is omitted for brevity

    GetCart(ctx, ownerID string) (domain.Cart, error)

    AddItem(ctx, ownerID string, item domain.CartItem) error

    DeleteItem(ctx, ownerID string, productID uuid.UUID) (bool, error)
}
```

Domain Cart

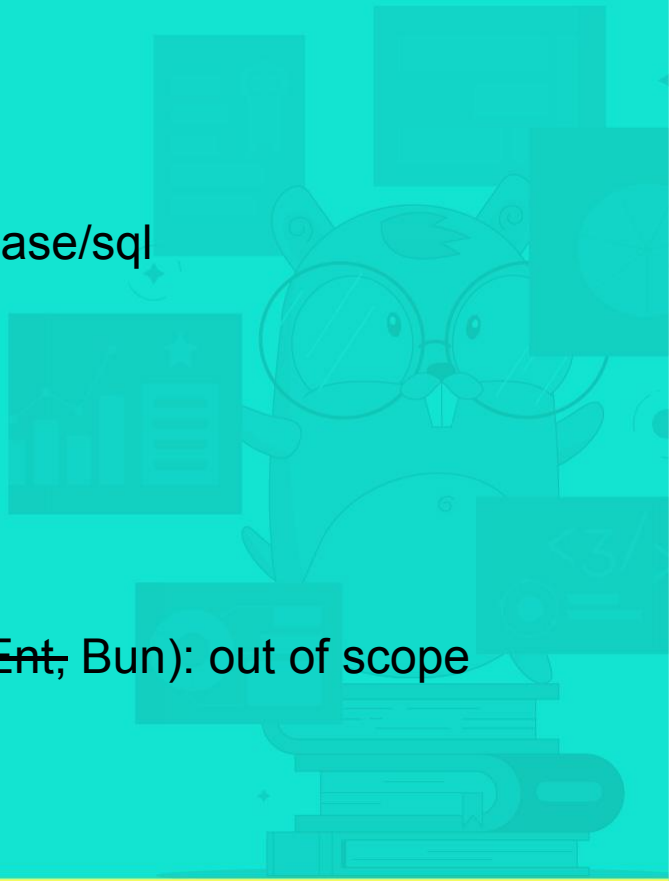
```
type Cart struct {
    OwnerID string
    Items   []CartItem
}

type CartItem struct {
    ProductID uuid.UUID // google/uuid
    Price     Money

    CreatedAt time.Time
}

type Money struct {
    Amount   decimal.Decimal   // shopspring/decimal
    Currency currency.Unit     // x/text/currency
}
```

- Cart items table

```
CREATE TABLE IF NOT EXISTS cart_items (

    owner_id        VARCHAR(255)                              NOT NULL,
    product_id      UUID                                      NOT NULL,
    price_amount    DECIMAL                                   NOT NULL,
    price_currency  VARCHAR(3)                                NOT NULL,

    created_at      TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,

    PRIMARY KEY (owner_id, product_id)
);
```

Working with Postgres in Go

- Pure stdlib database/sql
- squirrel + sqlx
- pgx driver API
- sqlc
- ORMs (~~GORM, Ent,~~ Bun): out of scope



Pure stdlib database/sql

- API: stdlib *database/sql*
- Driver: *lib/pq*
- Query builder: none
- Struct mapping: manual scanning

Pure stdlib database/sql example

```go
func (r *repo) GetCart(ctx context.Context, ownerID string) (domain.Cart, error) {

    // db *sql.DB
    rows, _ := r.db.QueryContext(ctx,
        `SELECT product_id, created_at FROM cart_items WHERE owner_id=$1`, ownerID)
    defer rows.Close()

    // scanning directly into domain model
    var items []domain.CartItem
    for rows.Next() {
        var item domain.CartItem
        _ = rows.Scan(&item.ProductID, &item.CreatedAt)
        items = append(items, item)
    }

    // handle rows.Err()

    return domain.Cart{OwnerID: ownerID, Items: items}, nil
}
```

Adopting squirrel and sqlx

- jmoiron/sqlx scans to record structs, which need to be mapped to domain models.

- *lib/pq* is maintenance mode and recommends using *jackc/pgx* instead

- API: stdlib *database/sql*
- Driver: *lib/pq*
- Query builder: *Masterminds/squirrel*
- Struct mapping: *jmoiron/sqlx*

Adopting squirrel and sqlx example

```go
func (r *repo) GetCart(ctx context.Context, ownerID string) (domain.Cart, error) {

    query, args, _ := sq.Select("product_id", "created_at").
        From("cart_items").
        Where(sq.Eq{"owner_id": ownerID}).
        PlaceholderFormat(sq.Dollar).ToSql()


    var dbItems []dbCartItem // local struct with `db` tags, aka DB records or models

    // dbx *sqlx.DB
    if err := r.dbx.SelectContext(ctx, &dbItems, query, args...); err != nil {
        return domain.Cart{}, err
    }

    var items []domain.CartItem
    // mapping of dbItems []dbCartItem to domain cart items


    return domain.Cart{OwnerID: ownerID, Items: items}, nil
}
```

Adopting pgx

pgx API advantages:

- Full Postgres feature support (batching, listen/notify, copy)

- Better type handling (JSON, arrays, UUID)

- Advanced connection pool - pgxpool

- API: *jackc/pgx* or *database/sql*
- Driver: *jackc/pgx*
- Query builder: *Masterminds/squirrel*
- Struct mapping: *pgx.CollectRows* method

Adopting pgx driver and API

```go
func (r *repo) GetCart(ctx context.Context, ownerID string) (domain.Cart, error) {

    query, args, _ := sq.Select("product_id", "created_at").
        ... // same as before


    // pool *pgxpool.Pool
    rows, _ := r.pool.Query(ctx, query, args...)


    // collect directly into domain model
    // pgx.CollectRows takes cares of closing rows
    items, _ := pgx.CollectRows(rows, func(row pgx.CollectableRow) (domain.CartItem, error) {
        var item domain.CartItem
        err := row.Scan(&item.ProductID, &item.CreatedAt)
        return item, err
    })


    return domain.Cart{OwnerID: ownerID, Items: items}, nil

}
```

Adopting sqlc

SQL-compiler advantages:

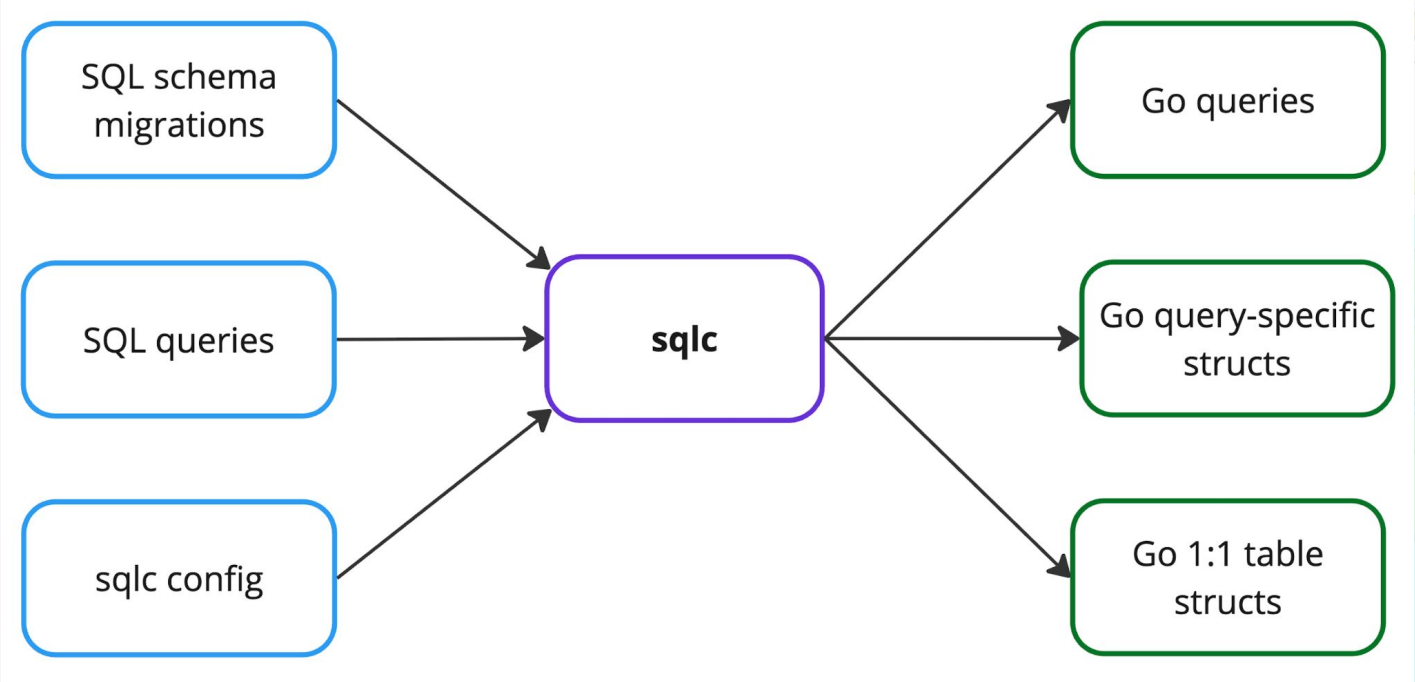
- Compile time safety: schema evolution, queries, type mapping

- Less boilerplate (no rows scanning)

- Separation of SQL and Go code: (queries are in .sql files)

- API: *jackc/pgx*
- Driver: *jackc/pgx*
- Queries: pure SQL
- Struct mapping: automatic to generated sqlc structs

sqlc in a nutshell



Generated queries and DB records

```
// Code generated by sqlc. DO NOT EDIT.
//   sqlc v1.29.0

const GetCart = `-- name: GetCart :many
SELECT product_id, created_at
FROM cart_items
WHERE owner_id = $1
`


type GetCartRow struct {
   ProductID string
   CreatedAt time.Time
}


func (q *Queries) GetCart(ctx context.Context, ownerID string)([]GetCartRow, error) {
    // generated implementation here
}
```

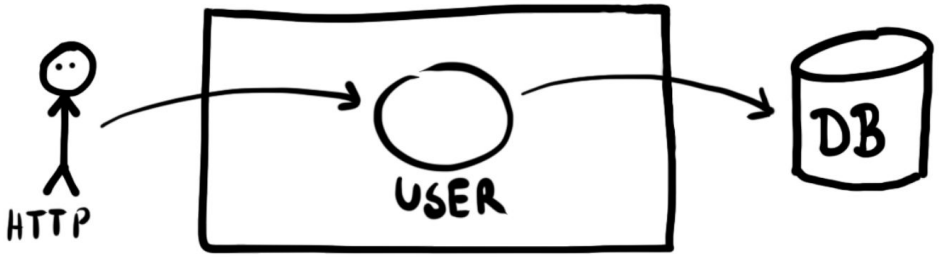
Delegating to sqlc generated queries

```go
func (r *repo) GetCart(ctx context.Context, ownerID string) (domain.Cart, error) {

    // rows []GetCartRow - generated by sqlc
    // q *db.Queries - generated by sqlc
    rows, _ := r.q.GetCart(ctx, ownerID)

    // map sqlc rows -> domain items
    items := mapGetCartRowsToDomain(rows)


    return domain.Cart{OwnerID: ownerID, Items: items}, nil

}
```

Generated sqlc structs vs domain models

- sqlc structs ≠ domain models
  - They reflect the database schema, not business concepts.
- Avoid using them in business logic
  - Otherwise, business logic would be tightly coupled to the persistence layer.
- Mapping between generated structs and domain models is **needed**

Single Model Couples Your Application



❌ **Anti-pattern: The Single Model**

Don't give a single model more than one responsibility.

Don't use more than one tag per structure field.

https://threedots.tech/post/common-anti-patterns-in-go-web-applications/

Repository methods vs generated sqlc queries

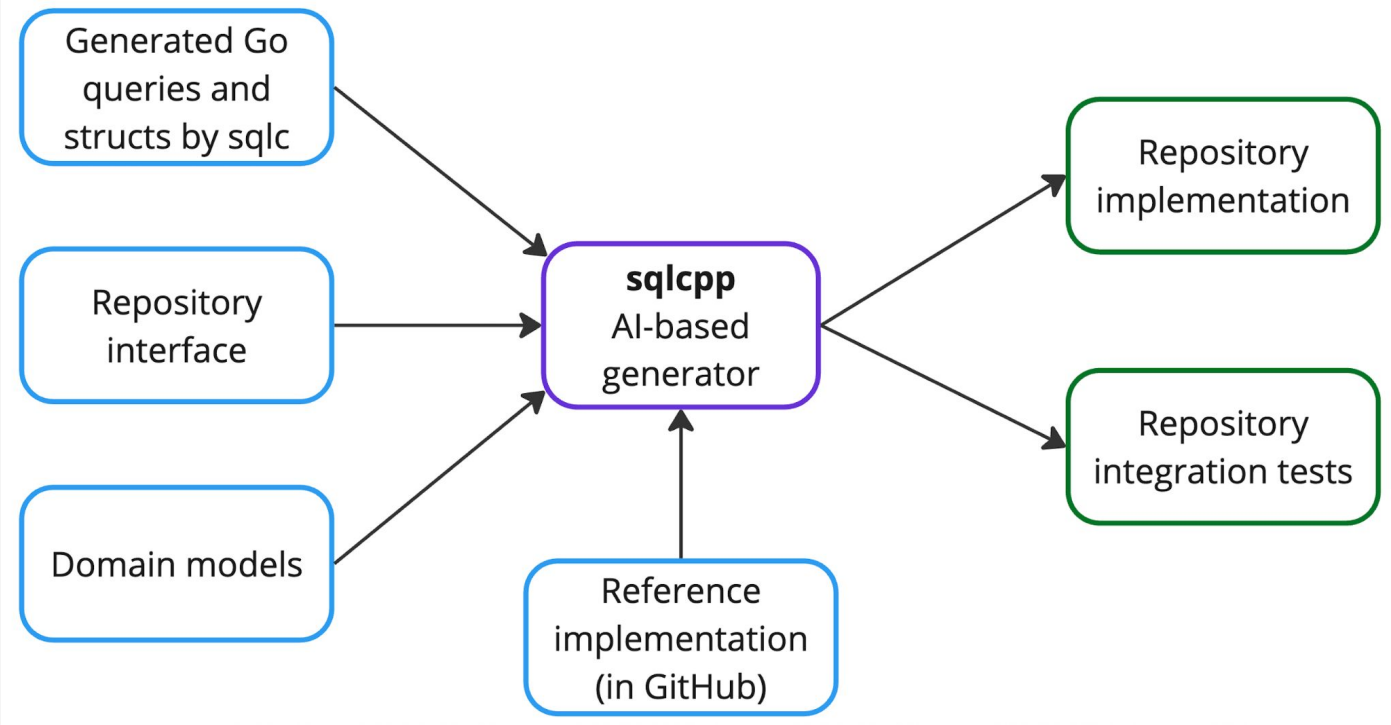
- Repository methods interface align with the **domain**, not SQL
- Repository methods compose multiple queries into a **transaction**
- Repository methods may add **tracing**, metrics, caching, etc

Introducing sqlc++

Let AI generate repository code by:

- Implementing provided (port) interface
- Using domain models and sqlc generated artifacts (queries and structs)
- Leveraging reference-based generation

In a nutshell



First version

- Single (huge) request to an LLM
- Uses *langchain-go* and *text/template*
- Fast generation (seconds), but inflexible

💡 Need to try Claude Code or Crush (ex-OpenCode)

Claude Code

- Agentic CLI tool powered by Anthropic models
- Specialized for coding tasks
- Can run shell commands
- Can refine results iteratively
- Extensible with custom commands (also hooks, MCPs)

Current version with Claude Code

- **Custom command** for Claude Code
- Multiple requests to an LLM, more expensive
- Slow generation (minutes), but flexible and powerful
- ***gh*** CLI tool to point to a reference implementation on GitHub

Claude Code's CLAUDE.md

- Similar to *README.md*, can be generated by */init* command
- Claude Code effectively adds it to every request to a LLM
- One can add instructions, i.e. how to fetch a GitHub file content:

```
- `gh api -H "Accept: application/vnd.github.raw"
  repos/nikolayk812/sqlcpp/contents/path/to/file.go`

  to fetch a GitHub file content from
 `https://github.com/nikolayk812/sqlcpp/blob/main/path/to/file.go`
```

Claude Code custom command

- <project_root>/**.claude/commands**/generate-repository.md

```
Generate a repository file for domain model: $ARGUMENTS

Follow these steps:

- Run `sqlc generate` command to generate files in `db` directory.

- Add generated files in `db` directory,`domain/$ARGUMENTS.go`
  and `port/$ARGUMENTS.go` to the context.

- Use this file from GitHub as a reference implementation:
  `github.com/nikolayk812/sqlcpp/blob/main/repository/order_repository.go`

- Create file `repository/$ARGUMENTS_repository.go`,
  it has to satisfy the port interface above.

- Make sure the generated file compiles and tests pass.
```

Reference implementation project

- Domain model: order
- 2 tables: *orders*, *order_items*
- Various operations: CRUD, search by a filter, soft-delete
- Various field types
  - uuid.UUID, decimal.Decimal, currency.Unit, time.Time, url.URL
  - UUID, DECIMAL, TIMESTAMP, JSON, JSONB, TEXT[]
- Transactions support

Demo time

Testing repositories

- Testcontainers + Postgres module
- Suite from *stretchr/testify*
- Table tests for each repository method
- Helpful libs: gofakeit, go-cmp

Test Suite

- Suite per a repository, single testcontainer per suite
- Start testcontainter in *SetupSuite* method
- Ryuk terminates unused containers if enabled, otherwise ➡️
- Stop testcontainer in *TearDownSuite* method

Test Suite

```
type cartRepositorySuite struct {
    suite.Suite

    repo      port.CartRepository
    pool      *pgxpool.Pool
}

func TestCartRepositorySuite(t *testing.T) {
    suite.Run(t, new(cartRepositorySuite))
}
```

Setup Suite

```
func (suite *cartRepositorySuite) SetupSuite() {
    ctx := suite.T().Context()

    connStr, err := startPostgres(ctx)
    suite.NoError(err)

    suite.pool, err = pgxpool.New(ctx, connStr)
    suite.NoError(err)

    suite.repo, err = repository.NewCart(suite.pool)
    suite.NoError(err)
}
```

Starting Postgres with Testcontainers

```
func startPostgres(ctx context.Context) (string, error) {

    container, err := postgres.Run(ctx, "postgres:17.6-alpine3.22",
        postgres.BasicWaitStrategies(),
        postgres.WithInitScripts("../migrations/01_cart_items.up.sql"),
    )
    // handle err

    connStr, err := container.ConnectionString(ctx, "sslmode=disable")
    // handle err

    return connStr, nil
}
```

Random test data with gofakeit

- less boilerplate, edge cases
- possible to pin specific fields for determinism

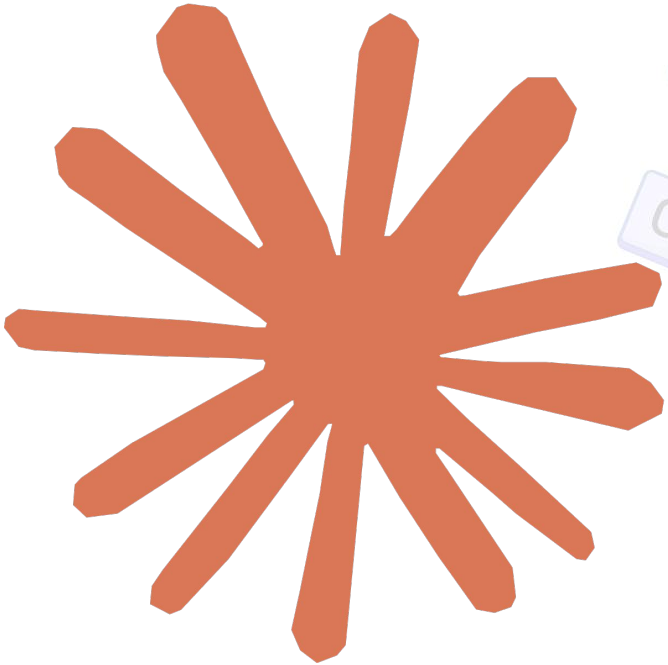
```
func randomCartItem() domain.CartItem {
    return domain.CartItem{
        ProductID: uuid.MustParse(gofakeit.UUID()),
        Price: domain.Money{
            Amount:   decimal.NewFromFloat(gofakeit.Price(1, 100)),
            Currency: currency.MustParseISO(gofakeit.CurrencyShort()),
        },
    }
}
```

Assertions with go-cmp

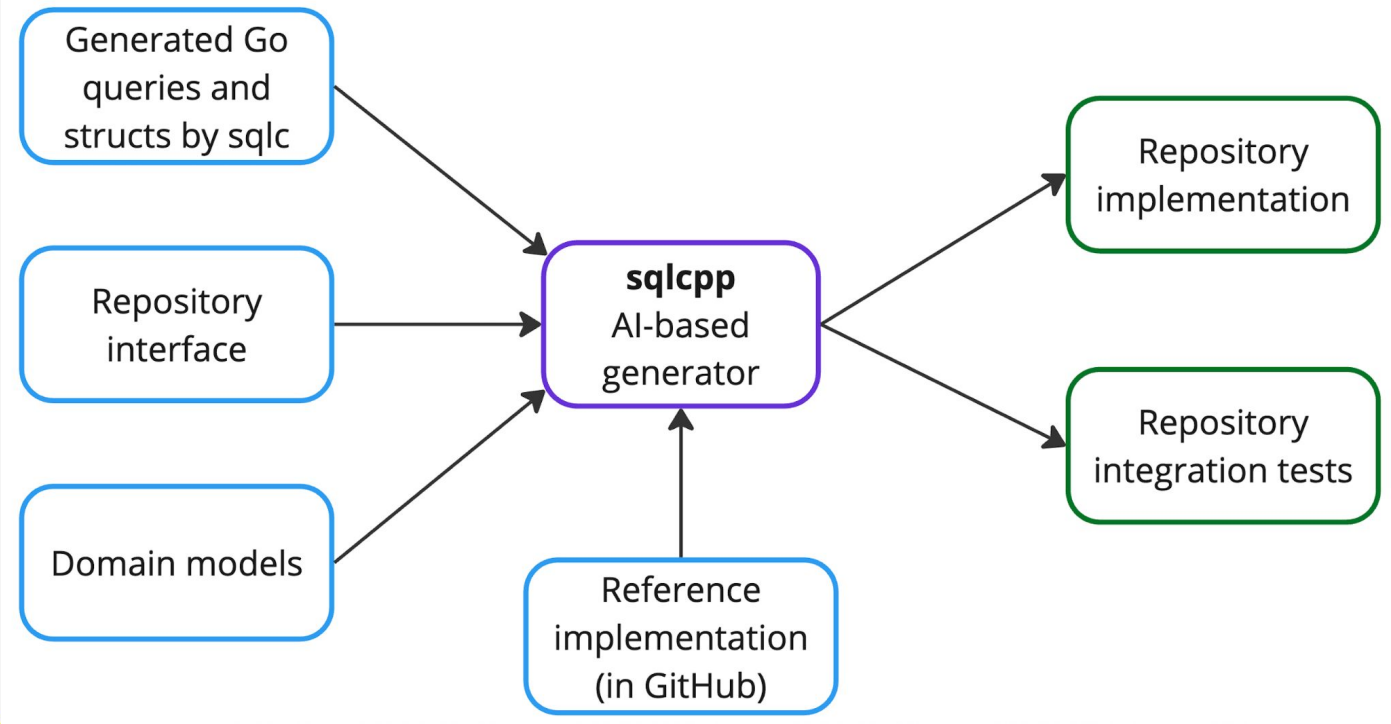
- readable diff, custom comparers, ignore fields

```
func assertCartItem(t *testing.T, expected, actual domain.CartItem) {
    t.Helper()

    currencyComparer := cmp.Comparer(func(x, y currency.Unit) bool {
        return x.String() == y.String()
    })

    opts := cmp.Options{
        cmpopts.IgnoreFields(domain.CartItem{}, "CreatedAt"),
        currencyComparer,
    }

    diff := cmp.Diff(expected, actual, opts)
    assert.Empty(t, diff)
}
```

Demo results



sqlc++ architecture recap



Takeaways

- Adopt sqlc, avoid use of generate structs in business logic
- Adopt agentic CLI tool: (Claude Code, Crush + GPT-OSS in Ollama)
- Reference-based generation provides more stable results
- Repository boilerplate is automated
- Schema, queries, and domain models remain manual

Thank you!

Q&A

Nikolay Kuznetsov

nikolayk812

nkuznetsov

github.com/nikolayk812/sqlcpp