avito.tech

# Transaction Management

## Repository and Unit of Work Patterns

Ilia Sergunin
Sr. Software Engineer

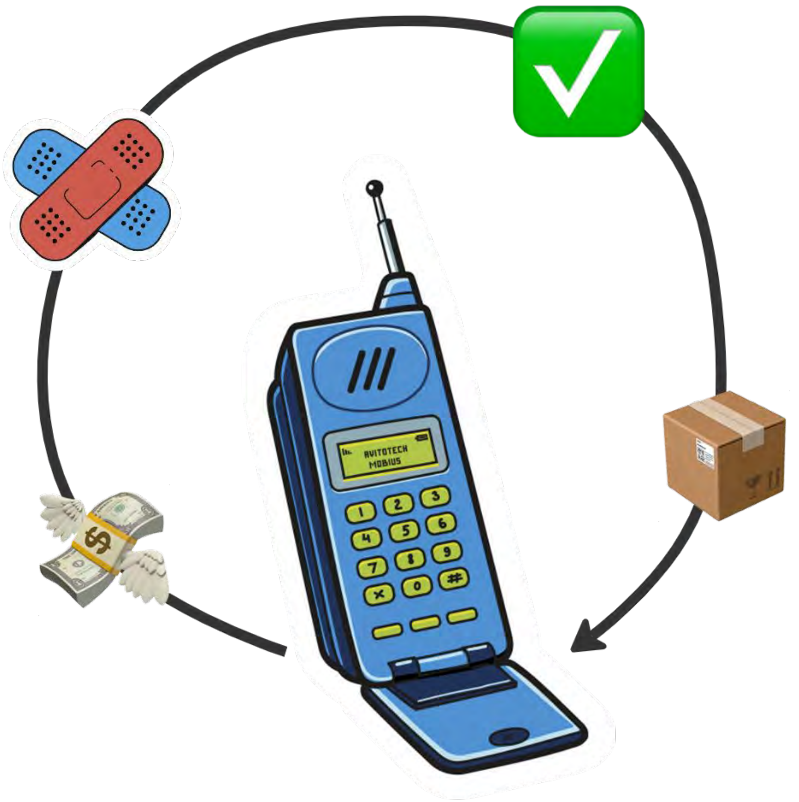
avito.tech

Avito

- Cloud & Network Services
- Command-line Interfaces (CLIs)
- DevOps & SRE
- Web Development

📈 ~300 M visits per month

## Smartphone reselling domain

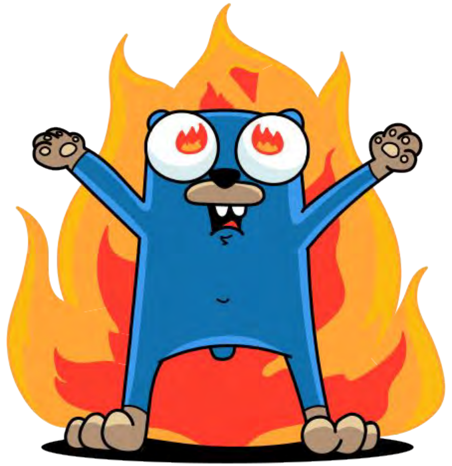


## Go Features

- Error Handling

## Error Handling

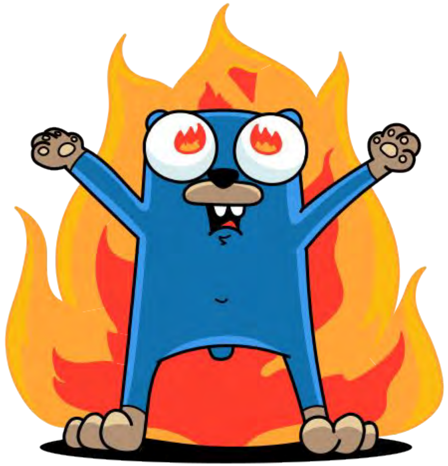
```go
res, err := someFunc()
if err ≠ nil {
    return err
}
```



## Error Handling

```go
res, err := someFunc()
~~if err != nil {~~
~~    return err~~
~~}~~
```

```go
res, err := someFunc()
```



avito.tech

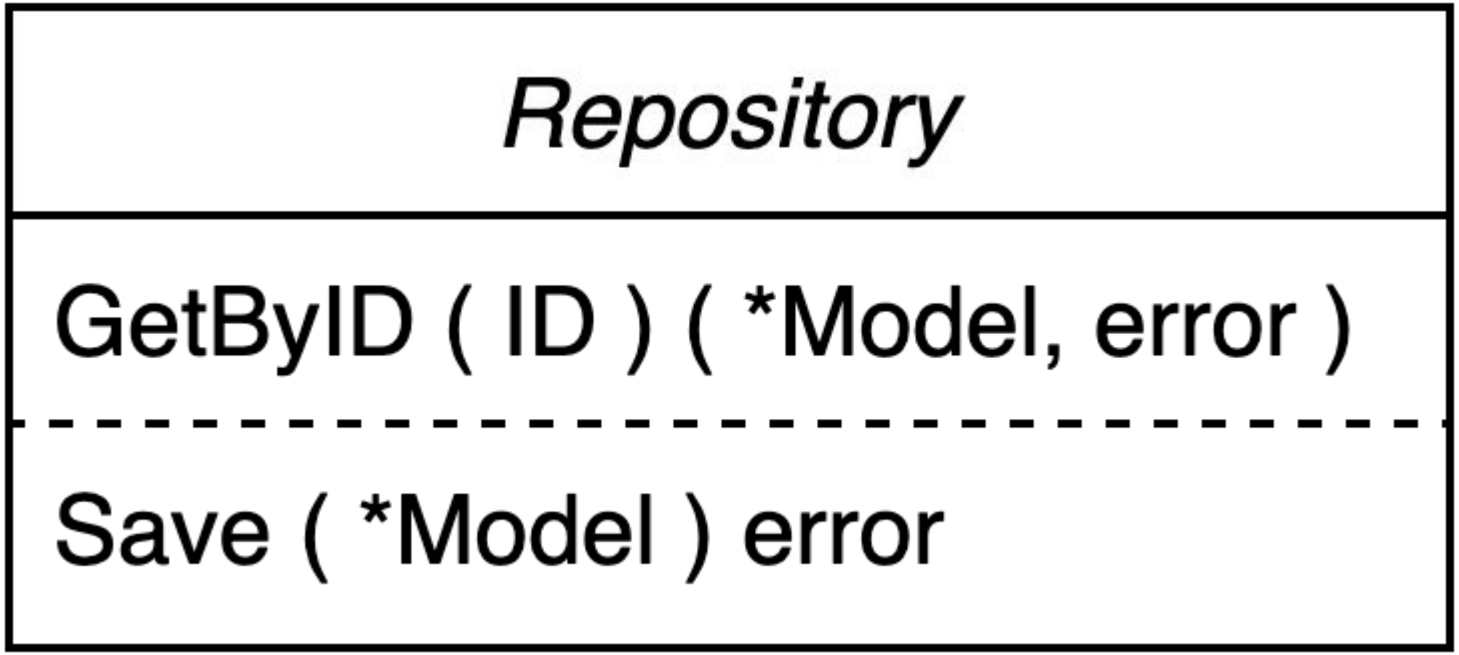
## Go Features

- Error Handling
- Not conventionally OOP

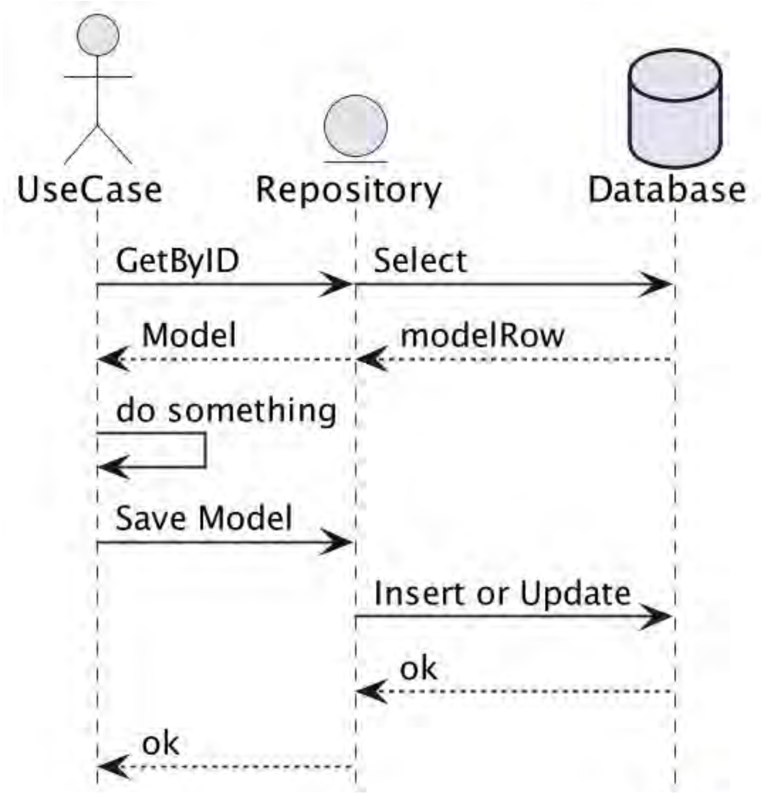
## Go Features

- Error Handling
- Not conventionally OOP
- Young and Developing

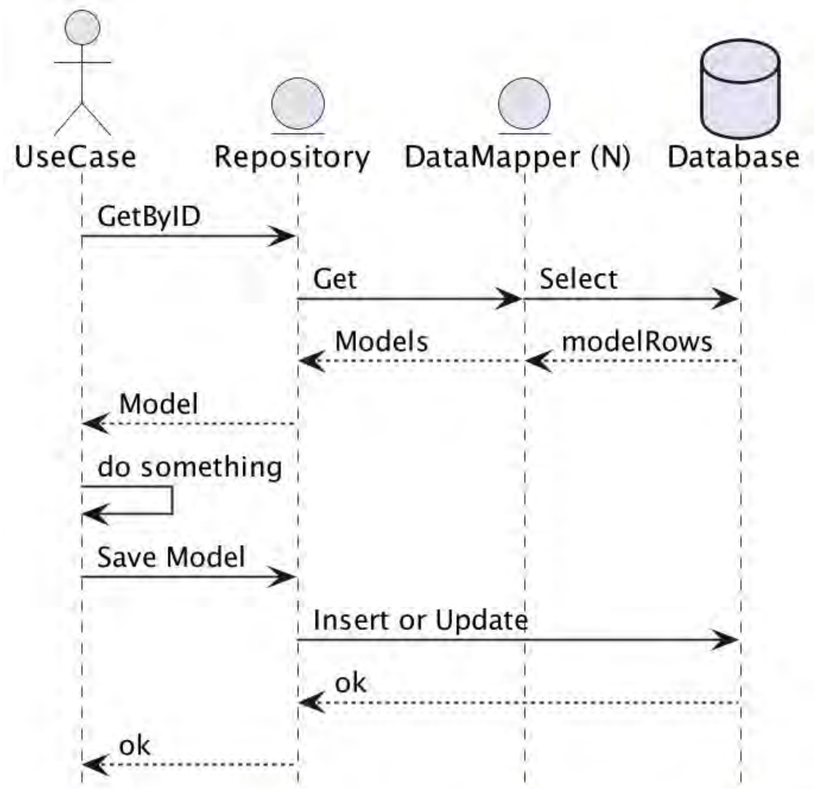
## Repository pattern



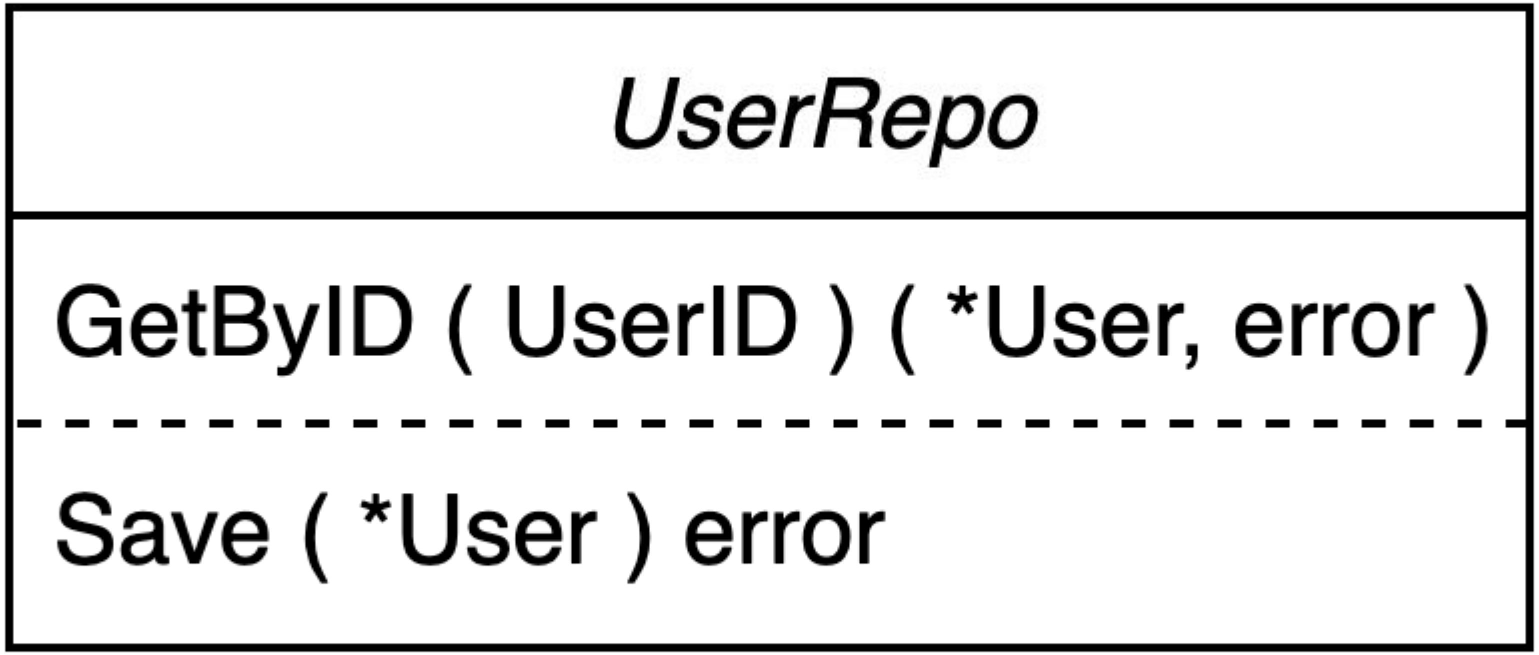
# Repository pattern



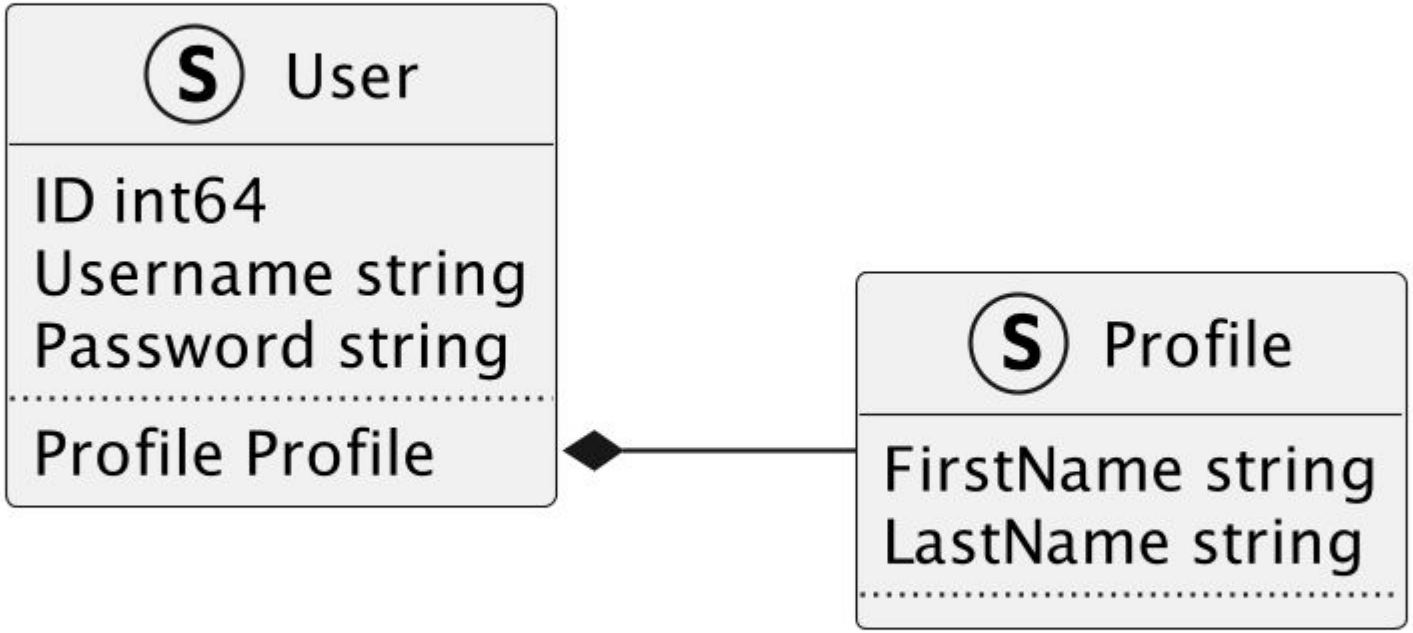
## Repository pattern



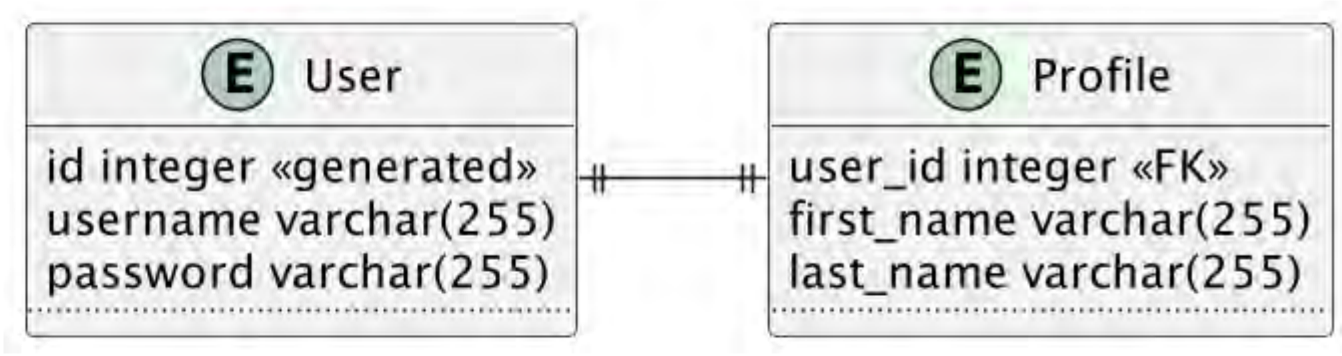
## User Repository



## User domain model



## User data in database



## Model

```go
type User struct {
    ID       UserID
    Username string
    Password string
}

type Profile struct {
    ID        ProfileID
    FirstName string
    LastName  string
}
```

## Row

```go
type userRow struct {
    ID         int64      `db:"id"`
    Username   string     `db:"username"`
    Password   string     `db:"password"`
    ProfileRow profileRow `db:"p"`
}

type profileRow struct {
    UserID    int64  `db:"user_id"`
    FirstName string `db:"first_name"`
    LastName  string `db:"last_name"`
}
```

## Getting of UserRepo

```
func (r *repo) GetByID(id UserID) (*User, error) {
   query := `SELECT u.*, p.user_id "p.user_id",
      p.first_name "p.first_name", p.last_name "p.last_name"
   FROM user u
        INNER JOIN profile p ON u.id = p.user_id
   WHERE u.id = ?;`

   uRow := userRow{}

   if err := r.db.Get(&uRow, r.db.Rebind(query), id); err ≠ nil {
       return nil, err
   }

   return r.toModel(uRow), nil
}
```

## Saving of UserRepo

```go
func (r *userRepo) Save(u *User) error {
   query := `BEGIN;
       INSERT INTO user ( ... ) VALUES ( ... ) ON CONFLICT (id)
           DO UPDATE SET  ...  RETURNING id;
       INSERT INTO profile ( ... ) VALUES ( ... ) ON CONFLICT
(user_id) DO UPDATE SET  ... ;
       COMMIT;`
    // …
}
```

## Saving of UserRepo

```go
func (r *userRepo) Save(u *User) error {
   query := `BEGIN;
       INSERT INTO user ( ... ) DO UPDATE SET  ...  RETURNING id;
       INSERT INTO profile ( ... ) DO UPDATE SET  ... ;
       COMMIT;`

   uRow, pRow := r.toRow(u)

   rows, err := r.db.Query(query, uRow, pRow)
   defer rows.Close()
   rows.Next()
   err = rows.Scan(&u.ID)
   return err
}
```

## Registration

```go
func (u *usecase) Register(username string) (*User, error) {
   if username == "" {
       return nil, errors.New("invalid username")
   }

   user := &User{Username: username}
   if err := u.userRepo.Save(user); err != nil {
       return nil, err
   }

   return user, nil
}
```

# Registration

```go
func (u *usecase) Register(username string) (*User, error) {
    if username == "" {
        return nil, errors.New("invalid username")
    }

    user := &User{Username: username}
    if err := u.userRepo.Save(user); err != nil {
        return nil, err
    }

    return user, nil
}
```

## Registration

```go
func (u *usecase) Register(username string) (*User, error) {
	if username == "" {
		return nil, errors.New("invalid username")
	}

	user := &User{Username: username}
	if err := u.userRepo.Save(user); err != nil {
		return nil, err
	}

	return user, nil
}
```

## Registration

```go
func (u *usecase) Register(username string) (*User, error) {
   // validation is hidden

   user := &User{Username: username}
   err := u.userRepo.Save(user) // error handling is hidden

   // error handling is hidden
   err = u.queue.Publish(UserRegistered{user.ID})

   return user, nil
}
```

## Registration

```go
func (u *usecase) Register(username string) (*User, error) {
   // validation is hidden

   user := &User{Username: username}
   err := u.userRepo.Save(user) // error handling is hidden

   // error handling is hidden
   err = u.queue.Publish(UserRegistered{user.ID})

   return user, nil
}
```

What happens
if the queue drops? 🤔

# Registration with Transaction

```go
func (u *usecase) Register(username string) (*User, error) {
   // validation is hidden

   tr, err := u.db.Begin() // error handling is hidden

   user := &User{Username: username}
   err = u.userRepo.Save(user) // error handling is hidden

   err = u.queue.Publish(UserRegistered{user.ID}) // error handling

   err = tr.Commit() // or tr.Rollback()

   return user, nil
}
```

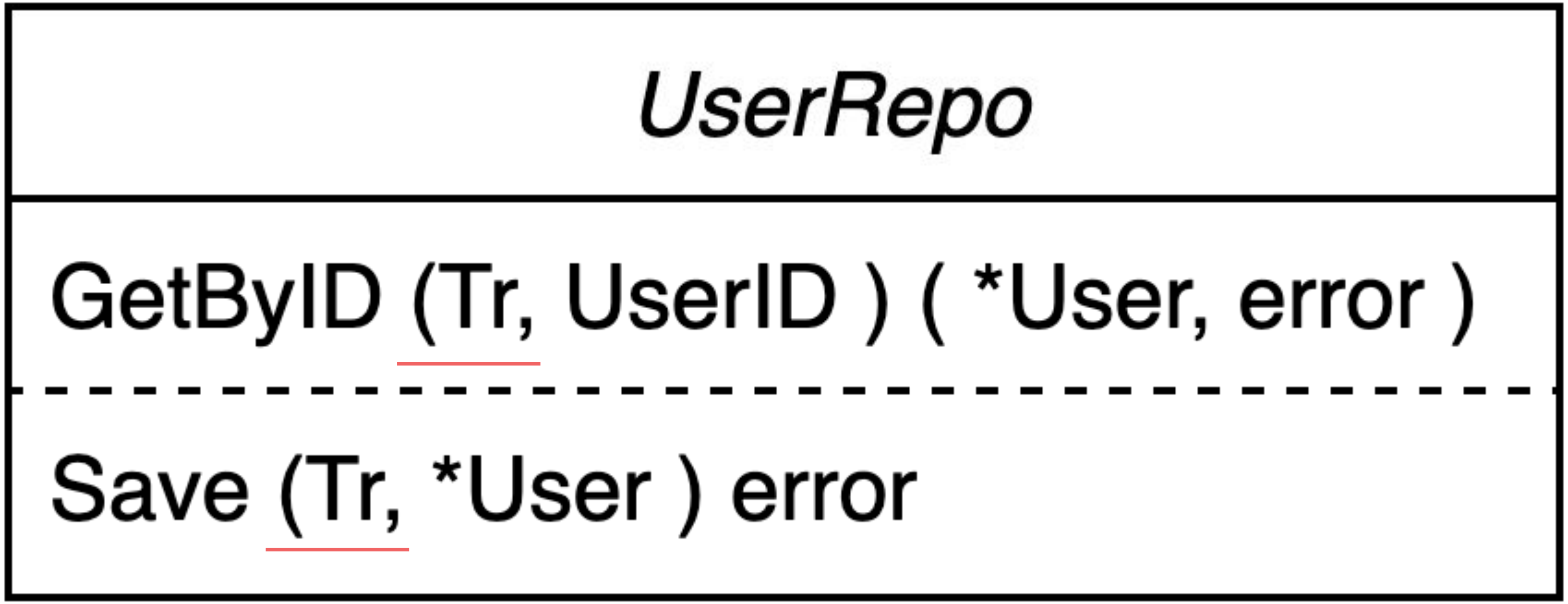
# Registration with Transaction

```go
func (u *usecase) Register(username string) (*User, error) {
    // validation is hidden

    tr, err := u.db.Begin()

    user := &User{Username: username}
    err = u.userRepo.Save(tr, user)

    err = u.queue.Publish(tr, UserRegistered{user.ID})

    err = tr.Commit() // or tr.Rollback()

    return user, nil
}
```

## Saving of UserRepo

```
func (r *userRepo) Save(tx *sqlx.Tx, u *User) error {
   query := `INSERT INTO user (username) VALUES (:username)
       ON CONFLICT (id)
           DO UPDATE SET username = EXCLUDED.username
       RETURNING id;`

   if tr == nil {
       tr = r.db
   }

   _, err := tr.Exec(query, args...)

   return err
}
```

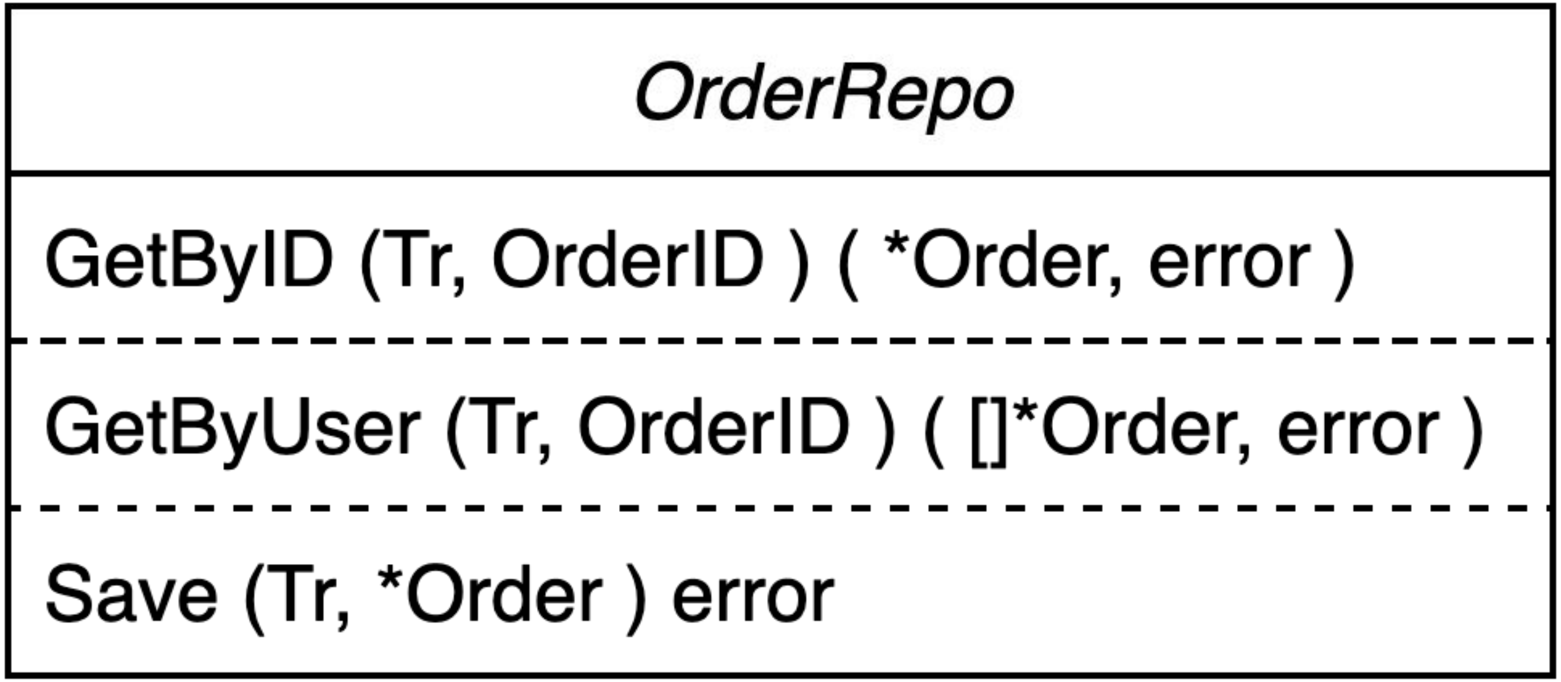
## User Repository


*UserRepo*

GetByID (Tr, UserID ) ( *User, error )

Save (Tr, *User ) error

## Order Repository


*OrderRepo*
GetByID (Tr, OrderID ) ( *Order, error )
GetByUser (Tr, OrderID ) ( []*Order, error )
Save (Tr, *Order ) error

## Buying

```go
func (u *usecase) Buy(uID UserID, pID ProductID, q int) (*Order, error) {
   // validation is hidden

   tr, err := u.db.Begin()

   order := &Order{UserID: uID, ProductID: pID, Quantity: q}
   err = u.orderRepo.Save(tr, order)
   err = u.queue.Publish(tr, Bought{order.ID})

   err = tr.Commit() // or tr.Rollback()

   return order, nil
}
```

## Fast Buy

```go
func (u *usecase) FastOrder(in In) error {
    user, err := u.Register(in.Username)

    _, err = u.Buy(user.ID, in.ProductID, in.Quantity)

    return nil
}
```

## Fast Buy with Transacion

```go
func (u *usecase) FastOrder(in In) error {
    tr, err := u.db.Begin()

    user, err := u.Register(tr, in.Username)

    _, err = u.Buy(tr, user.ID, in.ProductID, in.Quantity)

    err = tr.Commit() // or tr.Rollback()

    return nil
}
```

avito.tech

## Register

```
func (u *usecase) Register(username string) (*User, error) {
   // validation is hidden

   tr, err := u.db.Begin()



   // save to db and send to queue


   err = tr.Commit() // or tr.Rollback()

   return user, nil
}
```

## Register

```go
func (u *usecase) Register(tr *sqlx.Tx, username string) (*User, error) {
    hasExternalTransaction := true
    if tr == nil {
        tr, err := u.db.Begin()
        hasExternalTransaction = false
    }

    // save to db and send to queue

    if hasExternalTransaction {
        err = tr.Commit() // or tr.Rollback()
    }
    return user, nil
}
```

avito.tech

## What We Have

- 👍 Nested Transactional Use Cases
- ❌ Spreading of Knowledge about Transaction
- ❌ Duplication Code

# What We Want

- ❌ Ideal Repository
- ✅ Nested Transactional Use Cases
- ❌ Hide Transaction Control
- ❌ Database Replacement

## Closure in UserRepository

```go
func (r *userRepo) FastOrder(fn func() (*User, *Order, error)) error {
   tr, err := r.db.Begin()

   user, order, err := fn()    // use case execution

   err = r.Save(tr, user)
   err = r.orderRepo.Save(tr, order)

   err = tr.Commit() // or tr.Rollback()

   return nil
}
```

# Closure in UserRepository

```
func (r *userRepo) FastOrder(fn func() (*User, *Order, error)) error {
    tr, err := r.db.Begin()

    user, order, err := fn()

    err = r.Save(tr, user)
    err = r.orderRepo.Save(tr, order)

    err = tr.Commit() // or tr.Rollback()

    return nil
}
```

🤔 Does userRepo definitely need to know about Order?

## Closure

```go
func WithTransaction(tr *sqlx.Tx, fn func(*sqlx.Tx) error) error {
   hasExternalTransaction := true
   if tr == nil {
       tr, err = DB.Begin()
       hasExternalTransaction = false
   }

   err := fn(tr) // use case execution

   if hasExternalTransaction {
       err = tr.Commit() // or tr.Rollback()
   }
   return nil
}
```

## Register

```go
func (u *usecase) Register(tr *sqlx.Tx, username string) (*User, error) {
    hasExternalTransaction := true
    if tr == nil {
        tr, err := u.db.Begin()
        hasExternalTransaction = false
    }

    // save to db and send to queue

    if !hasExternalTransaction {
        err = tr.Commit() // or tr.Rollback()
    }
    return user, nil
}
```

## Register

```go
func (u *usecase) Register(tr *sqlx.Tx, username string) (*User, error) {
   user := &User{Username: username}
   err := WithTransaction(tr, func(tr *sqlx.Tx) error {
       err := u.userRepo.Save(tr, user)
       err = u.queue.Publish(tr, UserCreated{user.ID})
   })

   return user, err
}
```

## Register

```go
func (u *usecase) Register(tr *sqlx.Tx, username string) (*User, error) {
   user := &User{Username: username}
   err := WithTransaction(tr, func(tr *sqlx.Tx) error {
       err := u.userRepo.Save(tr, user)
       err = u.queue.Publish(tr, UserCreated{user.ID})
   })

   return user, err
}
```

## Factory Method in Repository

```go
type Tr interface {
   // *sqlx.DB and *sqlx.Tx
}

func NewRepo(tr Tr, log log.Logger) *userRepo {
   return &userRepo{
       tr:  tr,
       log: log,
   }
}

func (r *userRepo) WithTransaction(tr *sqlx.Tx) *userRepo {
   return NewRepo(tr, r.log)
}
```

## Saving of UserRepo

```
func (r *userRepo) Save(tx *sqlx.Tx, u *User) error {
   query := `INSERT INTO user (username) VALUES (:username)
       ON CONFLICT (id)
           DO UPDATE SET username = EXCLUDED.username
       RETURNING id;`

   if tr == nil {
       tr = r.db
   }

   _, err := tr.Exec(query, args...)

   return err
}
```

# Saving of UserRepo

```go
func (r *userRepo) Save(u *User) error {
   query := `INSERT INTO user (username) VALUES (:username)
       ON CONFLICT (id)
           DO UPDATE SET username = EXCLUDED.username
       RETURNING id;`

   _, err := r.tr.Exec(query, args... )

   return err
}
```

## Register

```
func (u *usecase) Register(tr *sqlx.Tx, username string) (*User, error)
{
   // validation is hidden
   user := &User{username}
   err := WithTransaction(tr, func(tr *sqlx.Tx) error {
       userRepo := u.userRepo.WithTransaction(tr)

       err := userRepo.Save(user)
       err = u.queue.Publish(tr, UserCreated{user.ID})

       return nil
   })
   return user, nil
}
```

# Closure with Reflection

```
func WithTransaction(tr *sqlx.Tx, fn interface{}) error {
   // opening a transaction

   repos, err := getReposFromAgrs(tr, fn)
   preparedFn, err := prepare(fn, tr, repos... )

   err := preparedFn()

   // closing a transaction

   return nil
}
```

## Closure with Reflection

```
func WithTransaction(tr *sqlx.Tx, fn interface{}) error {
	// opening a transaction

	repos, err := getReposFromAgrs(tr, fn)
	preparedFn, err := prepare(fn, tr, repos...)

	err := preparedFn()

	// closing a transaction

	return nil
}
```

## Closure with Reflection

```go
func WithTransaction(tr *sqlx.Tx, fn interface{}) error {
   // opening a transaction

   repos, err := getReposFromAgrs(tr, fn)
   preparedFn, err := prepare(fn, tr, repos...)

   err := preparedFn()

   // closing a transaction

   return nil
}
```

# Closure with Reflection

```go
func WithTransaction(tr *sqlx.Tx, fn interface{}) error {
   // opening a transaction

   repos, err := getReposFromAgrs(tr, fn)
   preparedFn, err := prepare(fn, tr, repos ... )

   err := preparedFn()

   // closing a transaction

   return nil
}
```

## Register with Reflection

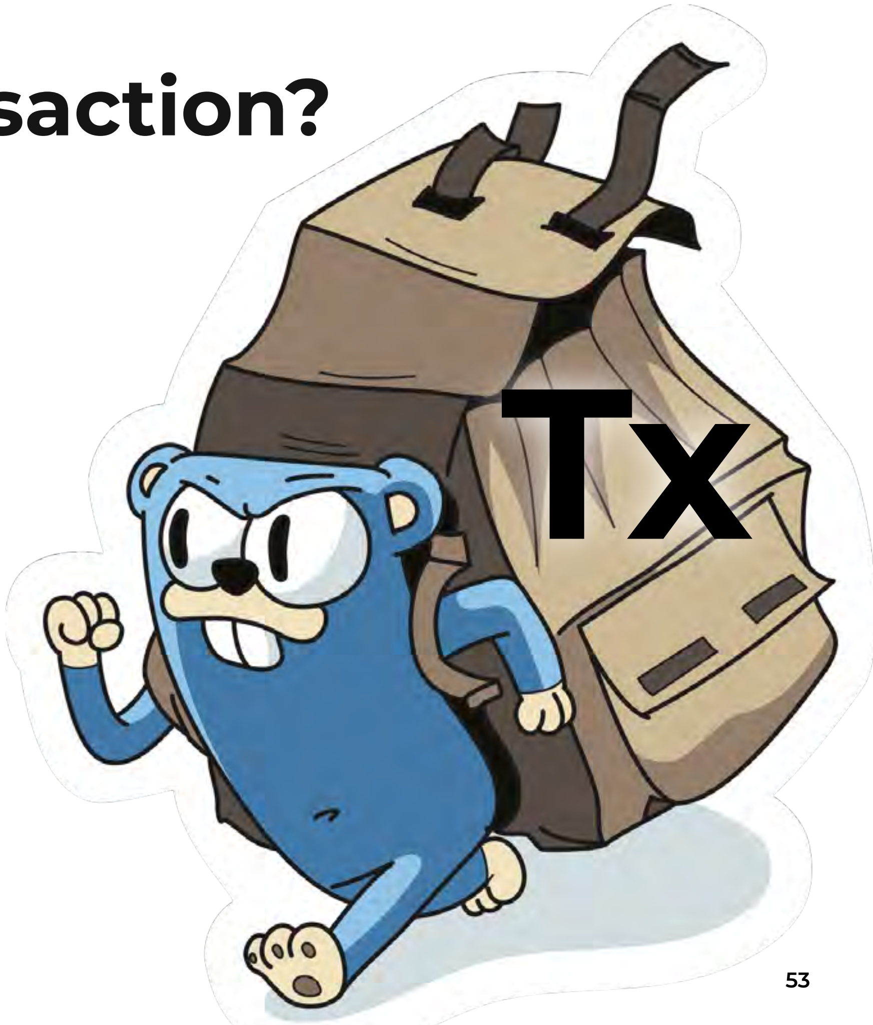
```go
func (u *usecase) Register(tr *sqlx.Tx, username string) (*User, error) {
   user := &User{Username: username}
   err := WithTransaction(tr, func(tr *sqlx.Tx, userRepo *UserRepo) error {
       err := u.userRepo.Save(user)
       err = u.queue.Publish(tr, UserCreated{user.ID})

       return nil
   })

   return user, nil
}
```

## Where to Store a Transaction?

- Passing as an Argument:
  - Python (SQLAlchemy)
- Global Variable:
  - PHP, JavaScript, Python (Django)
- Thread-Local Storage:
  - Java (Spring)
  - C# (system.Transaction)

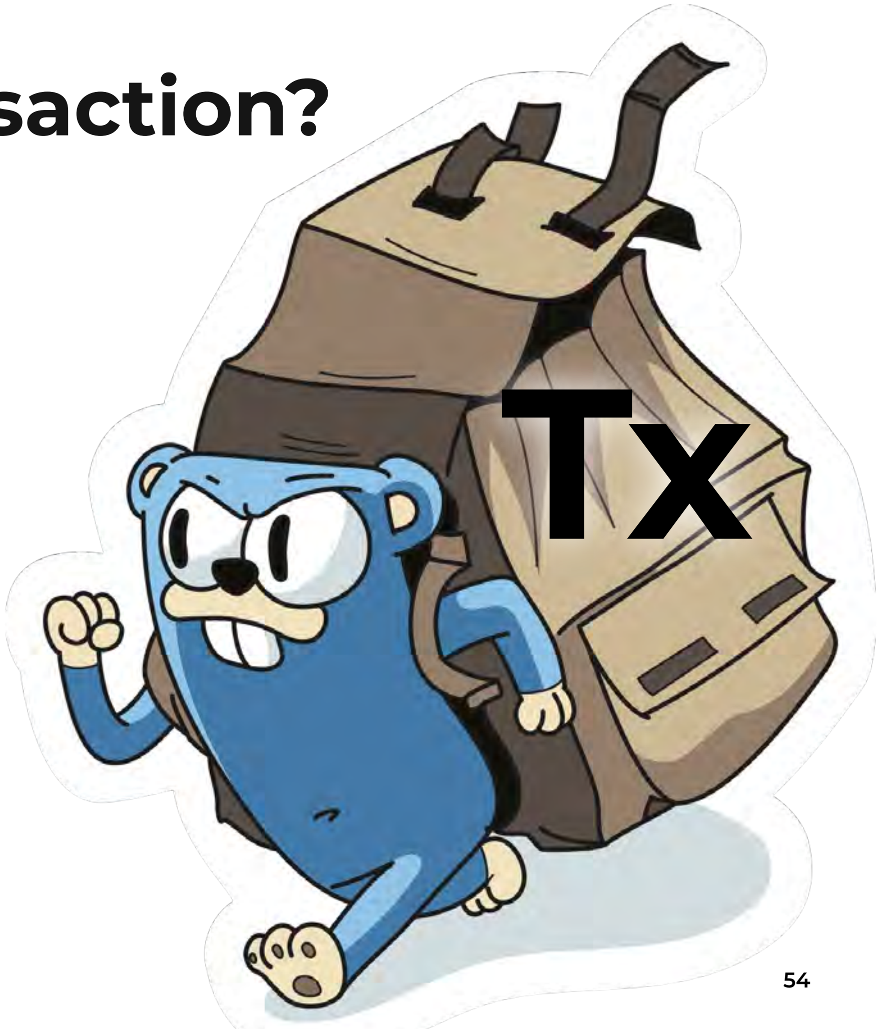
## Where to Store a Transaction?

- Passing as an Argument
- Local Storage
  - Based on Goroutine ID
  - Based on other hacks
- Context Package



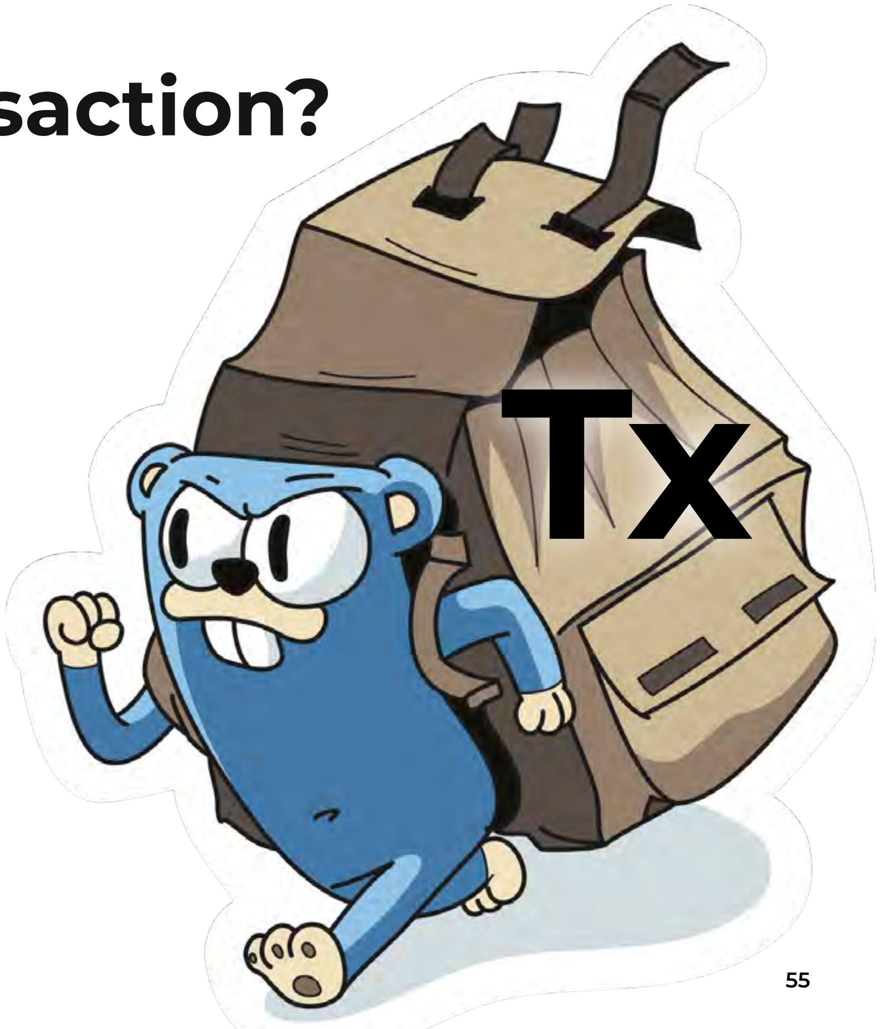
## Where to Store a Transaction?

- ~~Passing as an Argument~~
- Local Storage
  - Based on Goroutine ID
  - Based on other hacks
- Context Package



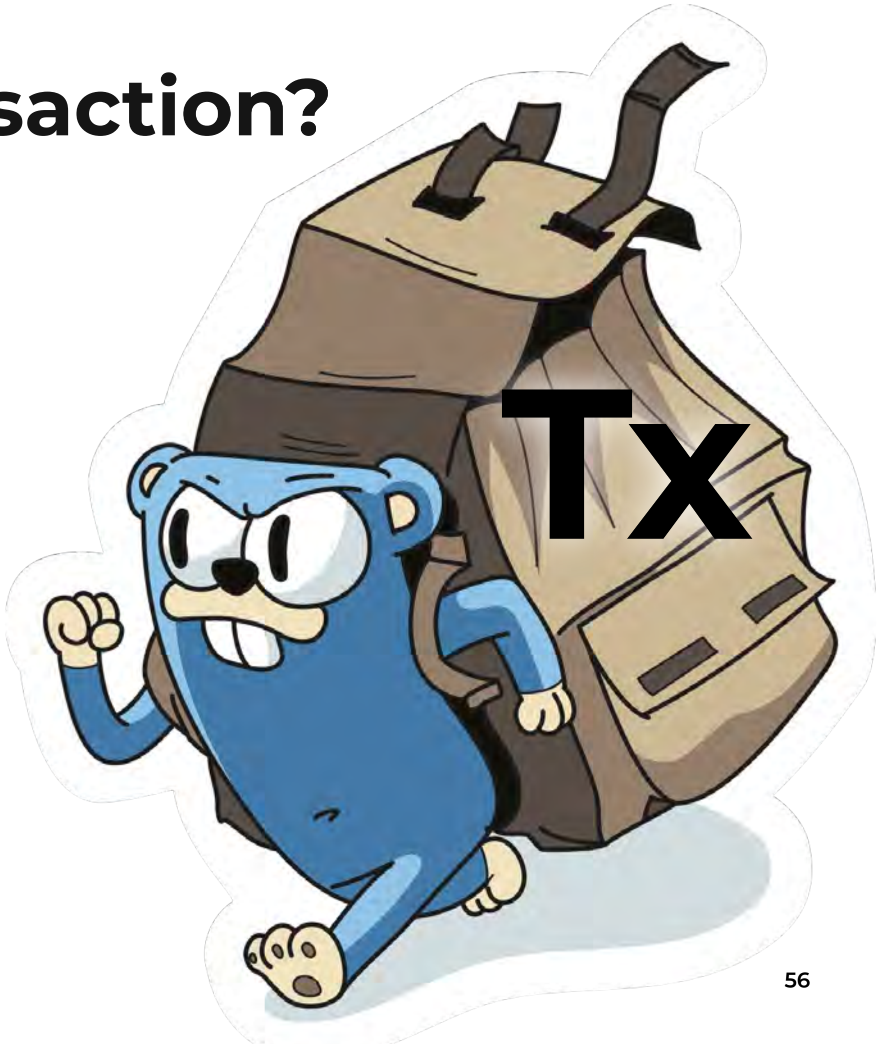
## Where to Store a Transaction?

- ~~Passing as an Argument~~
- Local Storage
  - ~~Based on Goroutine ID~~
  - Based on other hacks
- Context Package



## Where to Store a Transaction?

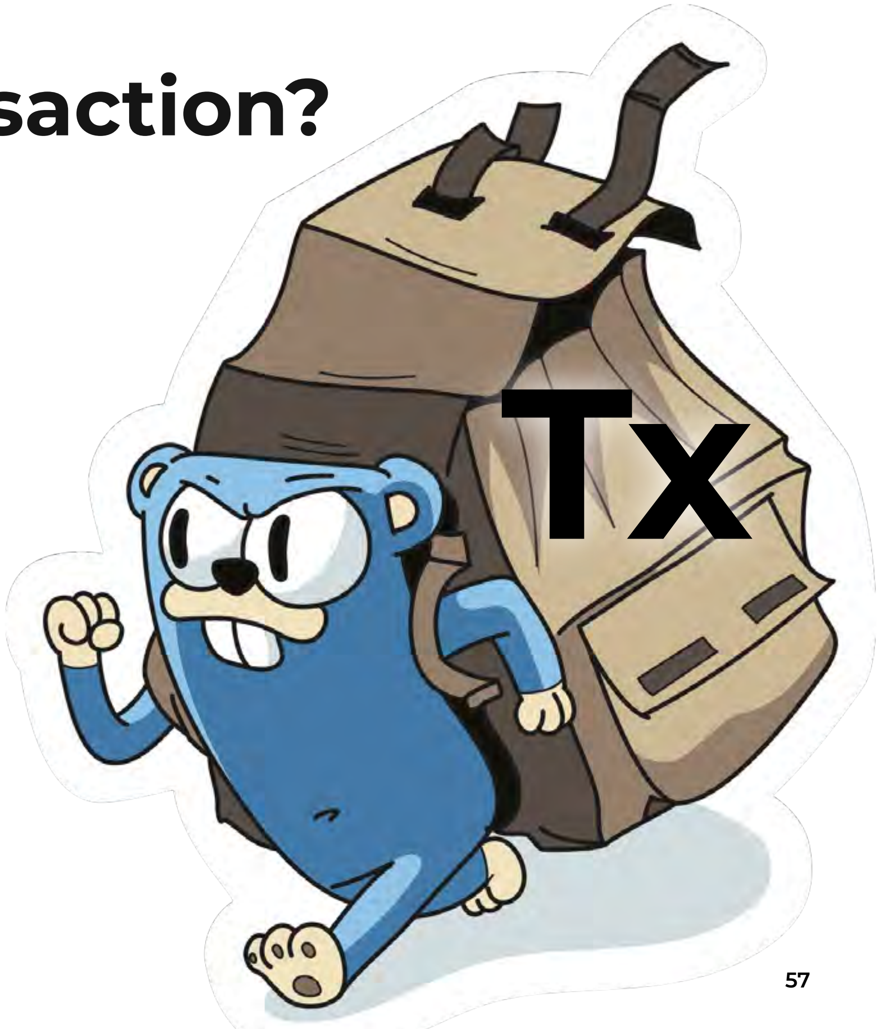
- ~~Passing as an Argument~~
- ~~Local Storage~~
  - ~~Based on Goroutine ID~~
  - ~~Based on other hacks~~
- Context Package



avito.tech

## Where to Store a Transaction?

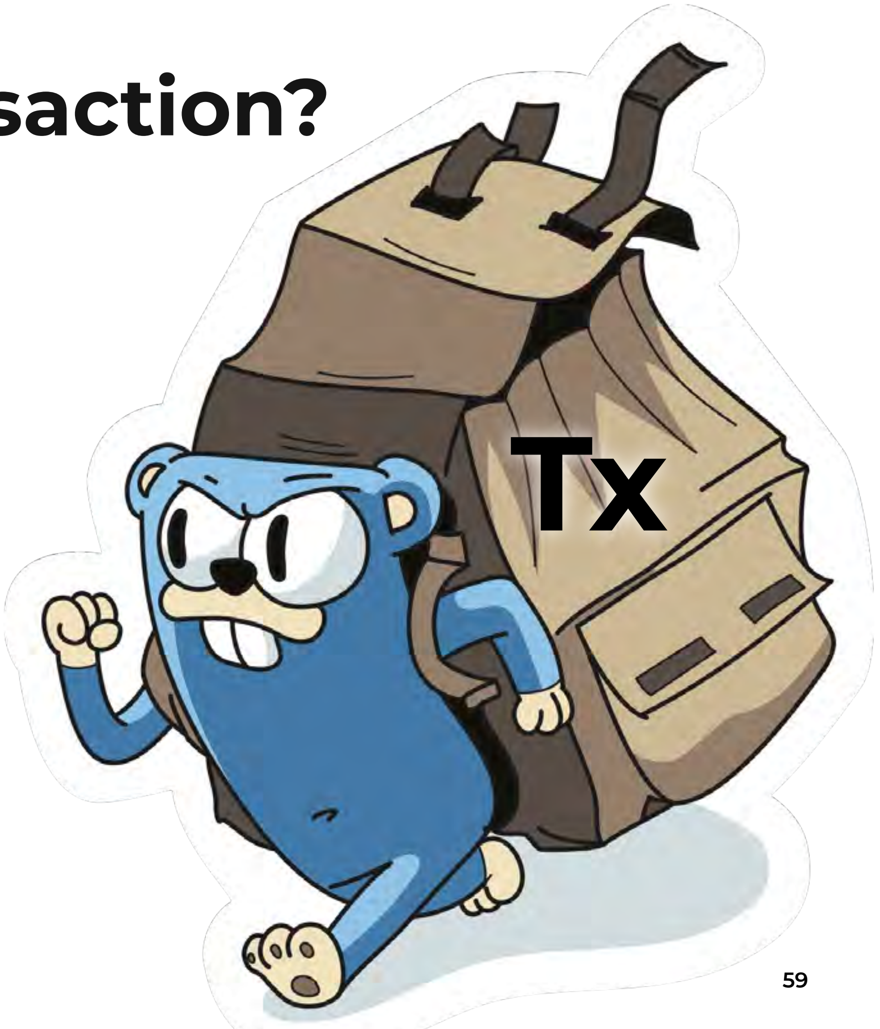
- ~~Passing as an Argument~~
- ~~Local Storage~~
  - ~~Based on Goroutine ID~~
  - ~~Based on other hacks~~
- ~~Context Package~~



## Where to Store a Transaction?

## Where to Store a Transaction?

- ~~Passing as an Argument~~
- ~~Local Storage~~
    - ~~Based on Goroutine ID~~
    - ~~Based on other hacks~~
- Context Package



## WithTransaction + Context

```go
func WithTransaction(ctx context.Context, fn func(context.Context) error) error {
   hasExternalTransaction := ctx.Value(ctxKey{}) != nil

   if !hasExternalTransaction {
        tr, err = DB.Begin()
        ctx = context.WithValue(ctx, ctxKey{}, tr)
   }

   err := fn(ctx) // call a usecase

   if !hasExternalTransaction {
        err = tr.Commit() // or tr.Rollback()
   }

   return nil
}
```

## Saving + Context

```go
func (r *userRepo) Save(ctx context.Context, u *User) error {
   query := `INSERT INTO user (username) VALUES (:username)
       ON CONFLICT (id)
           DO UPDATE SET username = EXCLUDED.username
       RETURNING id;`

   tr := r.db
   v, ok := ctx.Value(ctxKey{})
   if ok {
       tr = v.(Tr)
   }

   _, err := tr.Exec(query, u)

   return err
}
```

## What We Want

- ✅ Ideal Repository
- ✅ Nested Transactional Use Cases
- ✅ Hide Transaction Control
- ✅ Database Replacement

# What We Want

- ✅ Ideal Repository
- ✅ Nested Transactional Use Cases
- ✅ Hide Transaction Control
- ✅ Database Replacement
- ❌ Testable

## Transaction Manager

```go
type Manager interface {
   Do(context.Context, func(context.Context) error) error

   DoWithSettings(
     context.Context,
     Settings,
     func(context.Context) error,
   ) error
}
```

## Transaction

```go
type Transaction interface {
   IsActive() bool // defines the activity of a transaction
   Commit(context.Context) error // applies changes
   Rollback(context.Context) error // reverts changes
   Transaction() interface{} // returns the real transaction
}

// Creates a transaction
type TrFactory func(Settings) (Transaction, error)

// Creates a nested transaction if a database supports them
type NestedTrFactory interface {
   Begin(context.Context, Settings) (Transaction, error)
}
```

## Transaction Settings

```go
type Settings interface {
   // Combines two setting structures.
   EnrichBy(external Settings) Settings
   // Key to find the current transaction in Context.
   CtxKey() CtxKey
   // Sets up how to run transactions.
   Propagation() Propagation
   // Set flag of cancel the outer transaction by the nestedes.
   Cancelable() bool
   // Transaction execution timeout.
   TimeoutOrNil() *time.Duration
}
```

## Register

```go
func (u *usecase) Register(ctx context.Context, username string)
(*User, error) {
   // validation is hidden

   user := &User{Username: username}
   err := u.trm.Do(ctx, func(ctx context.Context) error {
       err := u.userRepo.Save(ctx, user)
       err = u.queue.Publish(ctx, UserCreated{user.ID})

       return err
   })

   return user, nil
}
```

## Register (Was)

```
func (u *usecase) Register(tr *sqlx.Tx, username string) (*User, error)
{
   hasExternalTransaction := true
   if tr == nil {
        tr, err := u.db.Begin()
        hasExternalTransaction = false
   }

   // save to db and send to queue

   if hasExternalTransaction {
        err = tr.Commit() // or tr.Rollback()
   }
   return user, nil
}
```

avito.tech

# Fast Buy

```
@Transaction
func (u *usecase) FastOrder(ctx context.Context, in In) error {
   user, err := u.Register(ctx, in.Username)

   _, err = u.Buy(ctx, user.ID, in.ProductID, in.Quantity)

   return err
}
```

## Fast Buy Decorator

```
type decorator struct{
   u *usecase
}

func (d *decorator) FastOrder(ctx context.Context, in In) error {
   return u.trm.Do(ctx, func(ctx context.Context) error {
       return d.usecase.FastOrder(ctx, in)
   })
}
```

## Generic Decorator

```go
type In struct {/* ... */}

type usecase struct {}
func (u *usecase) Handle(ctx context.Context, in In) error {
    // ...
}
```

## Generic Decorator

```go
type In struct {/* ... */}

type usecase struct {}
func (u *usecase) Handle(ctx context.Context, in In) error {
    // ...
}
```

avito.tech

## Generic Decorator

```go
type In struct {/* ... */}

type usecase struct {}
func (u *usecase) Handle(ctx context.Context, in In) error {
   // ...
}

type Usecase[In any] interface {
   Handle(ctx context.Context, in In) error
}
```

# Generic Decorator

```go
type txDecorator[In any] struct {
   manager Manager
   usecase Usecase[In]
}

func TxDecorate[In any](m Manager, u Usecase[In]) Usecase[In] {
   return &txDecorator[In]{manager: m, usecase: u}
}

func (d *txDecorator[In]) Handle(ctx context.Context, in In) (err error) {
   return d.manager.Do(ctx, func(ctx context.Context) error {
       return d.usecase.Handle(ctx, in)
   })
}
```

avito.tech

## Generic Decorator

```
usecase := TxDecorate(manager, usecase)


usecase.Handle(context.Background(), In{ /*..*/ })
```

avito.tech

## Getting Transaction

```
type CtxManager interface {
   Default(context.Context) Transaction
   ByKey(context.Context, CtxKey) Transaction
}
```

## Getting Transaction

```go
type CtxManager interface {
   Default(context.Context) Transaction
   ByKey(context.Context, CtxKey) Transaction
}

type Tr interface{/*  *sql.DB or *sql.Tx  */}

type SQLCtxManager interface {
   DefaultTrOrDB(context.Context, Tr) Tr
   TrOrDB(context.Context, CtxKey, Tr) Tr
}
```

avito.tech

## Saving of UserRepo (Was)

```
func (r *userRepo) Save(tx *sqlx.Tx, u *User) error {
   query := `INSERT INTO user (username) VALUES (:username)
       ON CONFLICT (id)
           DO UPDATE SET username = EXCLUDED.username
       RETURNING id;`

   if tr == nil {
       tr = r.db
   }

   _, err := tr.Exec(query, args...)

   return err
}
```

# Saving of UserRepo (Now)

```go
func (r *userRepo) Save(ctx context.Context, u *User) error {
   query := `INSERT INTO user (username) VALUES (:username)
       ON CONFLICT (id)
           DO UPDATE SET username = EXCLUDED.username
       RETURNING id;`

   _, err := r.getter.DefaultTrOrDB(ctx, r.db).
       ExecContext(ctx, query, args ... )

   return err
}
```

## What We Have

- ✅ Ideal Repository
- ✅ Nested Transactional Use Cases
- ✅ Hide Transaction Control
- ✅ Database Replacement
- ✅ Testable

## What Did It Cost?

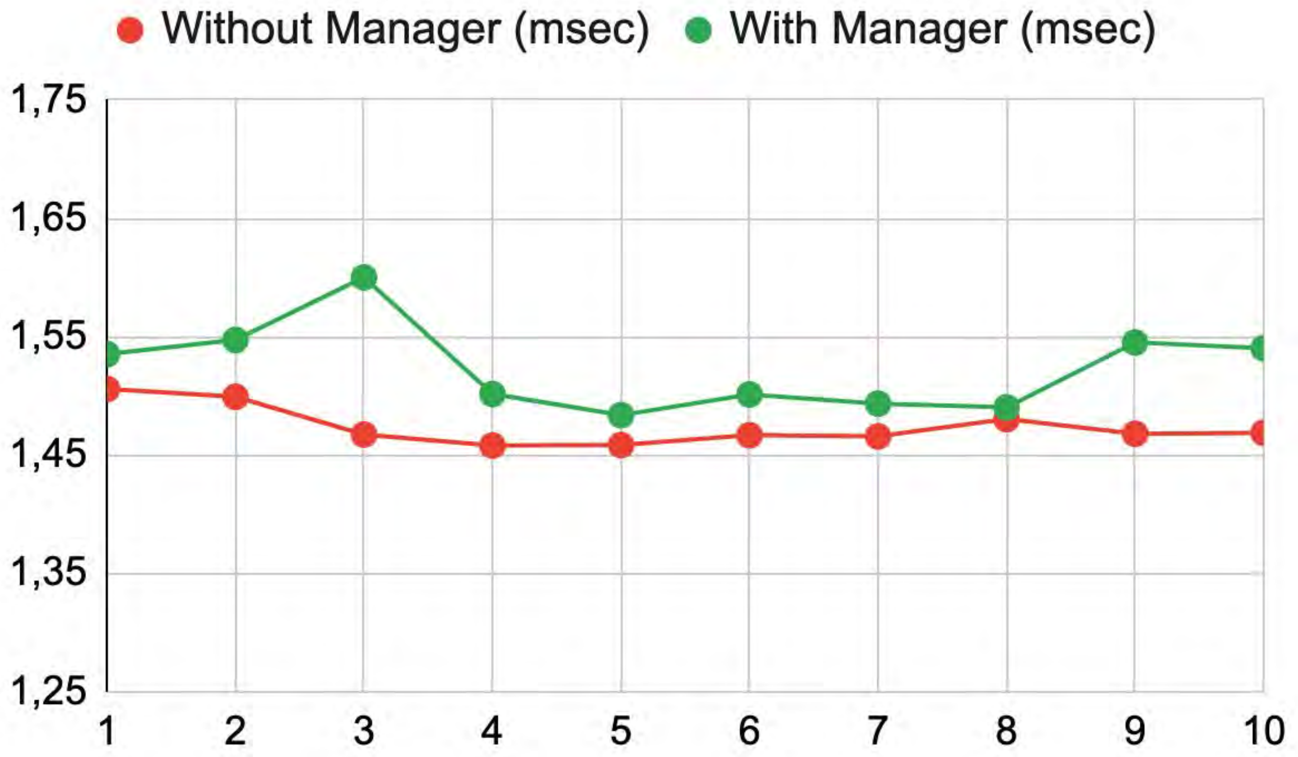
## Go >= 1.13

- Errors в 1.13
- Smooth update minor version In Go

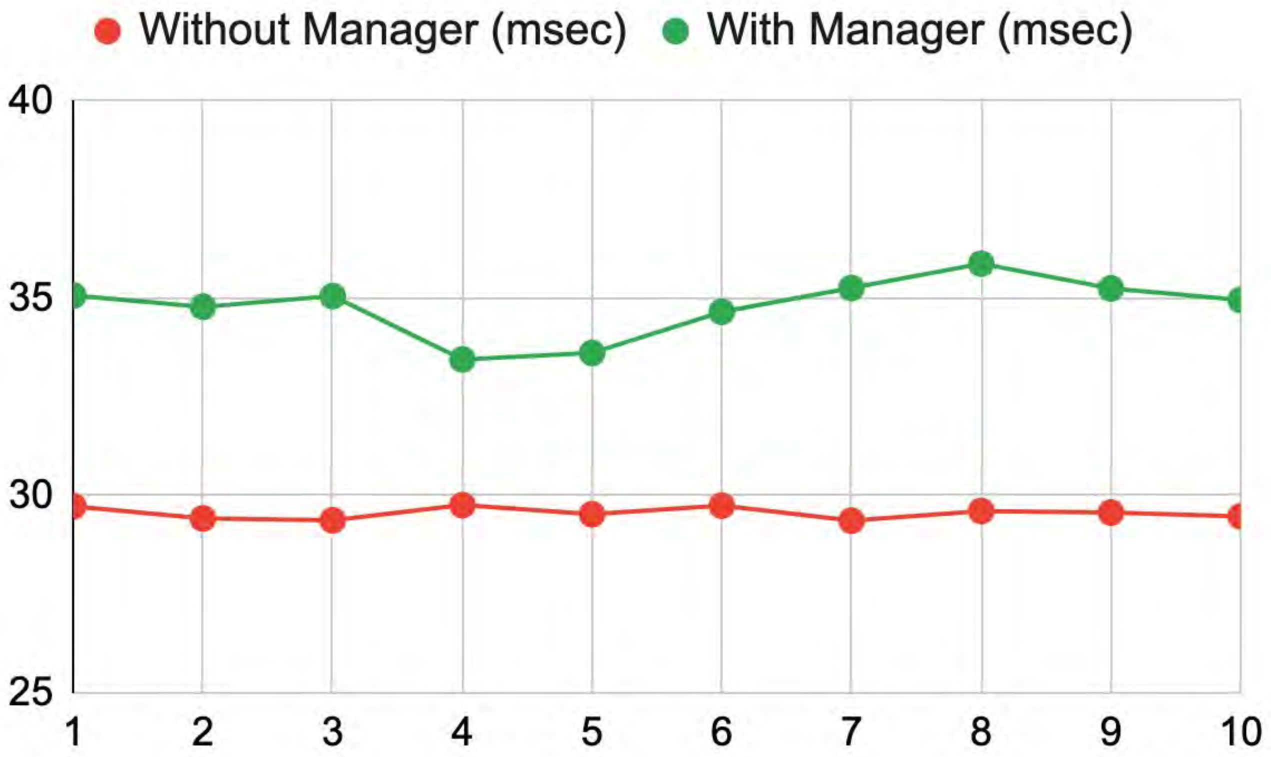
# A Few Adaptors

- database/sql
- jmoiron/sqlx
- gorm
- mongo-go-driver
- go-redis

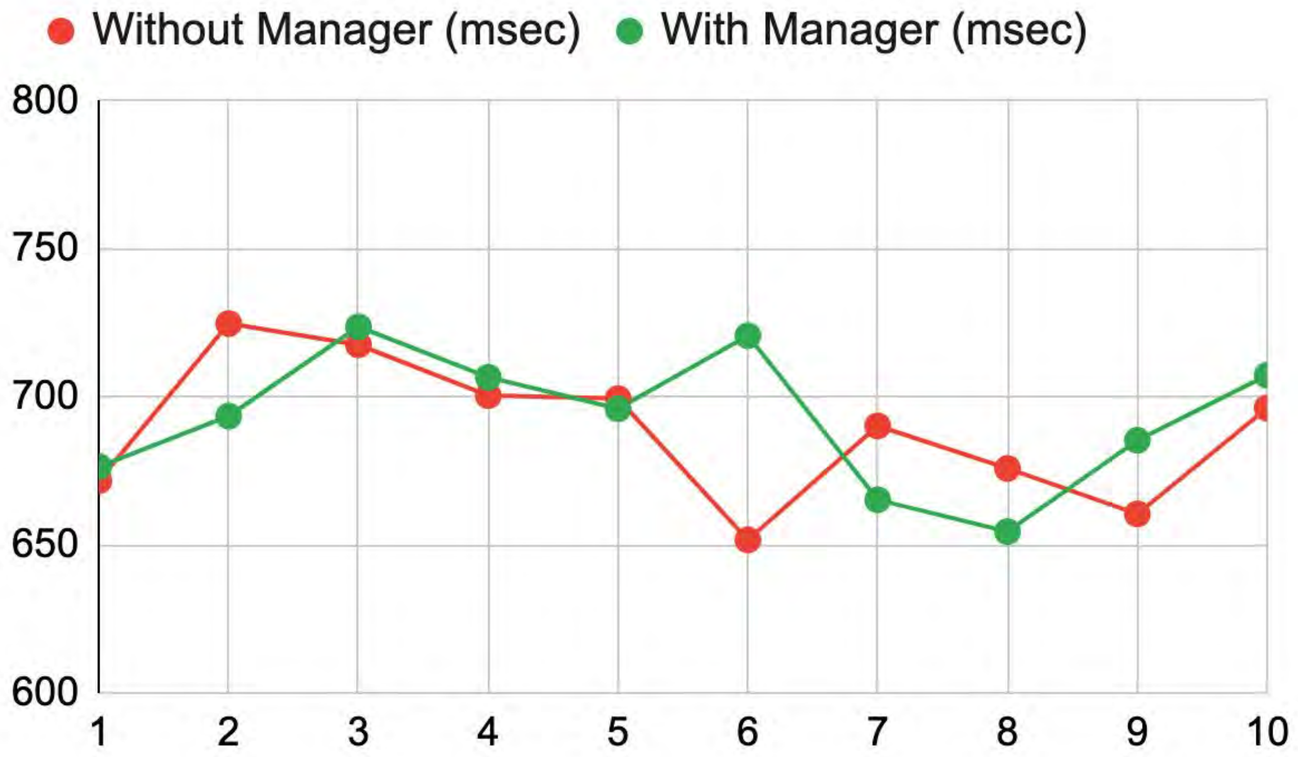
# Benchmark with SQLMock



# Benchmark with in Memory SQLite



## Benchmark with in Filesystem MySQL



## Carry Context Everywhere

```go
func (h *handler) Handle(w http.ResponseWriter, req *http.Request) {
    ctx := req.Context()
    err := h.trm.Do(ctx, func(ctx context.Context) error {
        // Top use case
        err = h.repo1.Save(ctx, model1)
        err = h.trm.Do(ctx, func(ctx context.Context) error {
            // Nested use case
            err = h.repo2.Save(ctx, model2)
        })
    })
    fmt.Fprintf(w, "done")
}
```

## Long Business Transaction

```go
func (u *usecase) Register(ctx context.Context, in In) (*User, error) {
   user := &User{Username: in.UN, passport: in.PP, phone: in.Ph}

   err := u.trm.Do(ctx, func(ctx context.Context) error {
       err = u.passportCheck(ctx, user)
       err = u.phoneCheck(ctx, user)

       err := u.userRepo.Save(ctx, user)
       err = u.queue.Publish(ctx, UserCreated{user.ID})

       return err
   })
   return user, nil
}
```

## Drawbacks

- Only works on Go >= 1.13
- A few adapters for ORMs
- ~17% performance drop (5 microseconds)
- Need to carry context everywhere
- Long transactions are not supported

## Long Business Transaction

**Was:**

```
type UserRepo interface {
   GetByID(*sqlx.Tx, UserID) (*User, error)
   Save(*sqlx.Tx, *User) error
}
```

**Now:**

```
type UserRepo interface {
   GetByID(context.Context, UserID) (*User, error)
   Save(context.Context, *User) error
}
```

# Saving of UserRepo (Was)

```go
func (r *userRepo) Save(tx *sqlx.Tx, u *User) error {
   query := `INSERT INTO user (username) VALUES (:username)
       ON CONFLICT (id)
           DO UPDATE SET username = EXCLUDED.username
       RETURNING id;`

   if tr == nil {
       tr = r.db
   }

   _, err := tr.Exec(query, args...)

   return err
}
```

## Saving of UserRepo (Now)

```go
func (r *userRepo) Save(ctx context.Context, u *User) error {
	query := `INSERT INTO user (username) VALUES (:username)
		ON CONFLICT (id)
			DO UPDATE SET username = EXCLUDED.username
		RETURNING id;`

	_, err := r.getter.DefaultTrOrDB(ctx, r.db).
		ExecContext(ctx, query, args...)

	return err
}
```

## Register (Was)

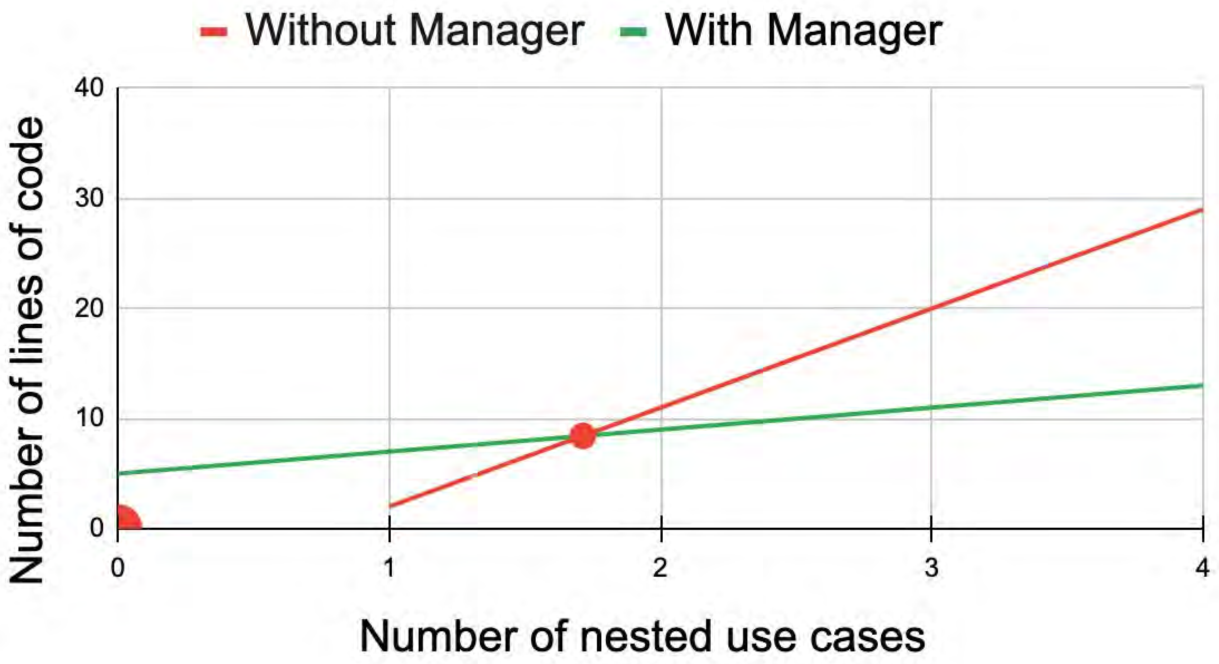
```go
func (u *usecase) Register(tr *sqlx.Tx, username string) (*User, error) {
	hasExternalTransaction := true
	if tr == nil {
		tr, err := u.db.Begin()
		hasExternalTransaction = false
	}

	// save to db and send to queue

	if !hasExternalTransaction {
		err = tr.Commit() // or tr.Rollback()
	}
	return user, nil
}
```

avito.tech

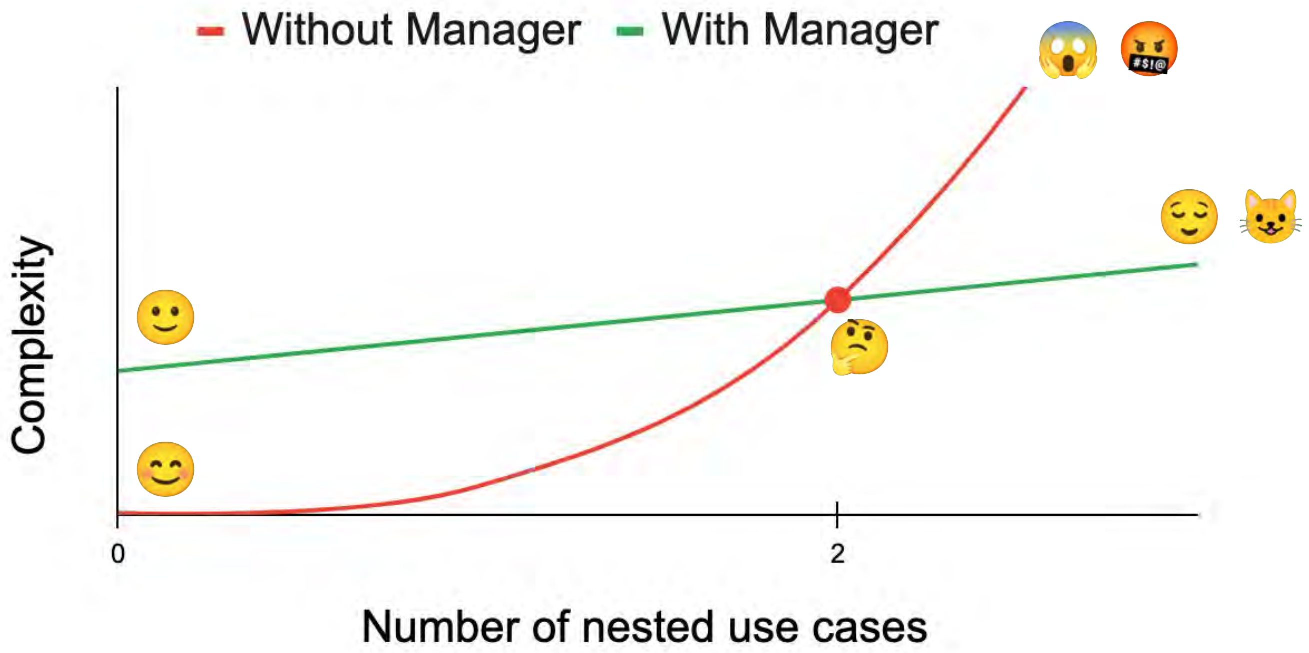
# Register (Now)

```go
func (u *usecase) Register(ctx context.Context, username string)
(*User, error) {
   // validation is hidden

   user := &User{Username: username}
   err := u.trm.Do(ctx, func(ctx context.Context) error {
       err := u.userRepo.Save(ctx, user)
       err = u.queue.Publish(ctx, UserCreated{user.ID})

       return err
   })

   return user, nil
}
```

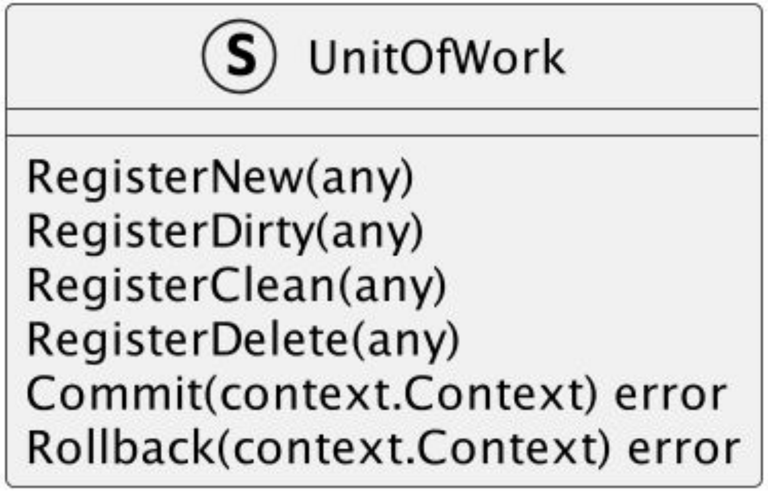
# Do You Have more 2 nested use cases?



# Do You Have more 2 nested use cases?

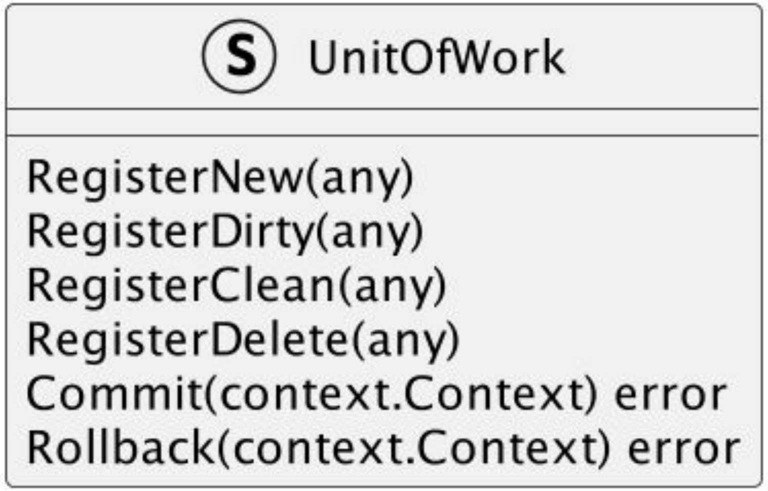


## Unit Of Work



```
RegisterNew(any)
RegisterDirty(any)
RegisterClean(any)
RegisterDelete(any)
Commit(context.Context) error
Rollback(context.Context) error
```

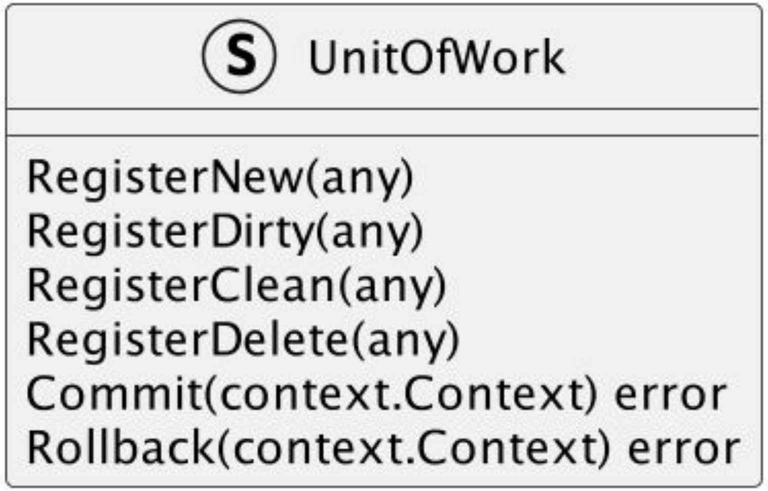
# Unit Of Work



RegisterNew(any)
RegisterDirty(any)
RegisterClean(any)
RegisterDelete(any)
Commit(context.Context) error
Rollback(context.Context) error

*Maintains a list of objects affected by a business transaction and coordinating the writing out of changes and the resolution of concurrency problems.*

## Unit Of Work


S UnitOfWork

RegisterNew(any)
RegisterDirty(any)
RegisterClean(any)
RegisterDelete(any)
Commit(context.Context) error
Rollback(context.Context) error

*Maintains a list of objects affected by a business transaction and coordinating the writing out of changes and the resolution of* ***concurrency problems****.*

## Advantages of UoW

- Batch changes

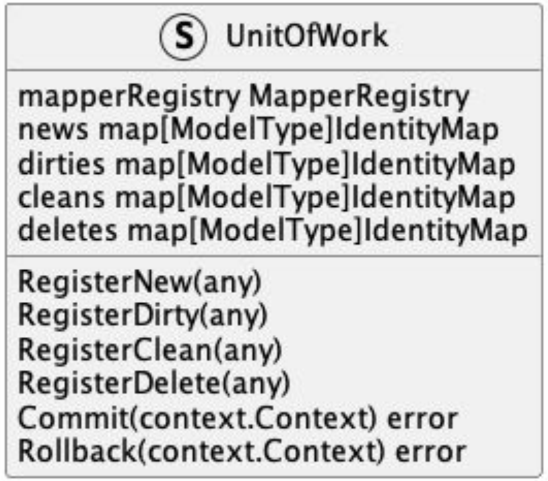
## Advantages of UoW

- Batch changes
- Long business transaction

## Disadvantages of UoW

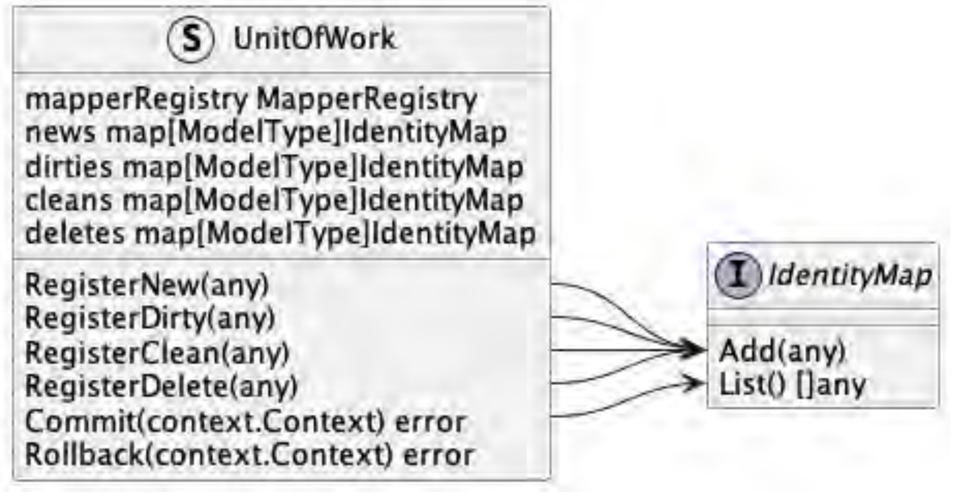
- Cannot use a Pessimistic Lock
- Complexity

# Class Diagram of UoW



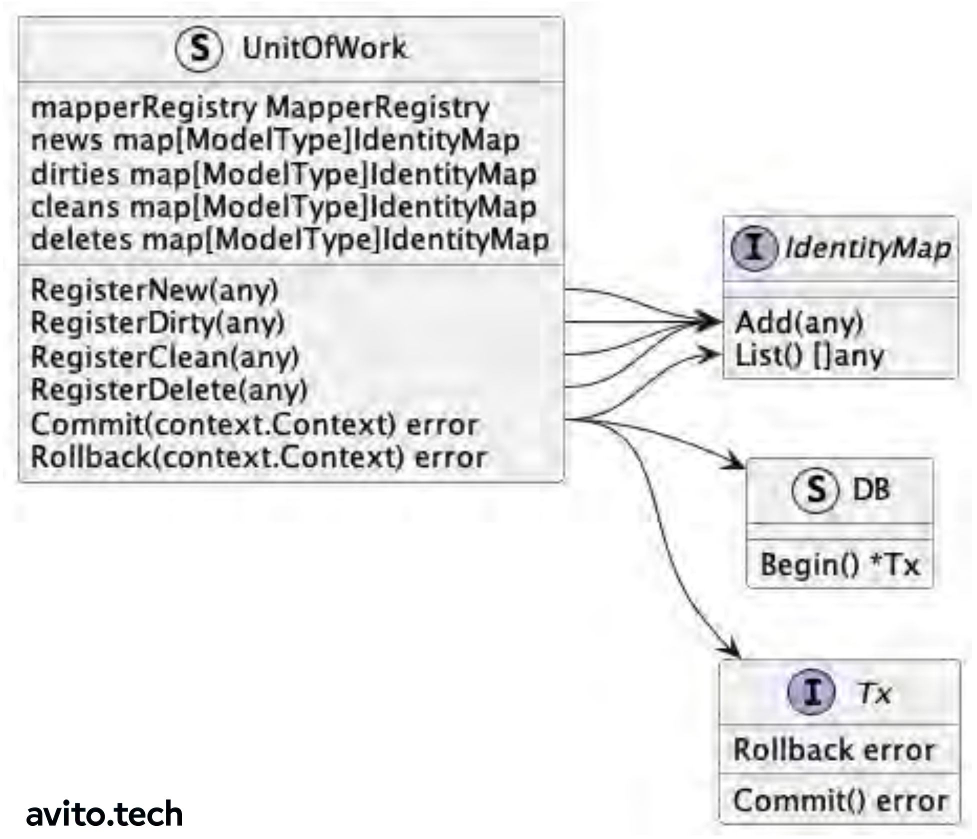
```
mapperRegistry MapperRegistry
news map[ModelType]IdentityMap
dirties map[ModelType]IdentityMap
cleans map[ModelType]IdentityMap
deletes map[ModelType]IdentityMap
RegisterNew(any)
RegisterDirty(any)
RegisterClean(any)
RegisterDelete(any)
Commit(context.Context) error
Rollback(context.Context) error
```

# Class Diagram of UoW

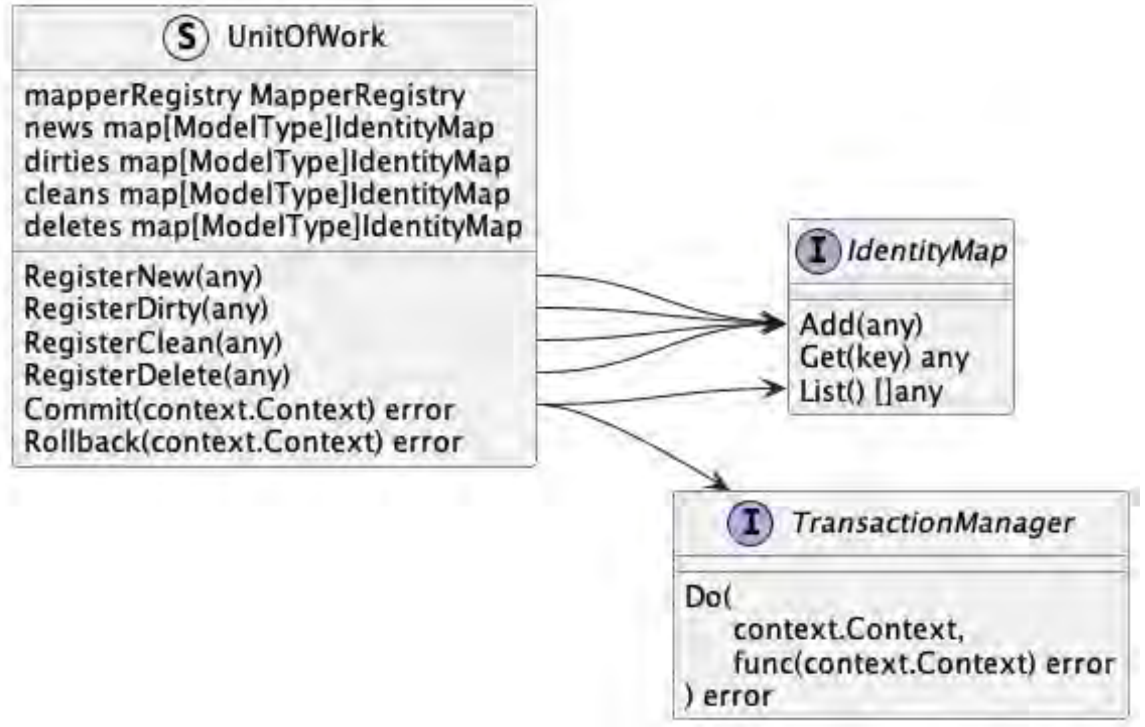


UnitOfWork
mapperRegistry MapperRegistry
news map[ModelType]IdentityMap
dirties map[ModelType]IdentityMap
cleans map[ModelType]IdentityMap
deletes map[ModelType]IdentityMap
RegisterNew(any)
RegisterDirty(any)
RegisterClean(any)
RegisterDelete(any)
Commit(context.Context) error
Rollback(context.Context) error
IdentityMap
Add(any)
List() []any

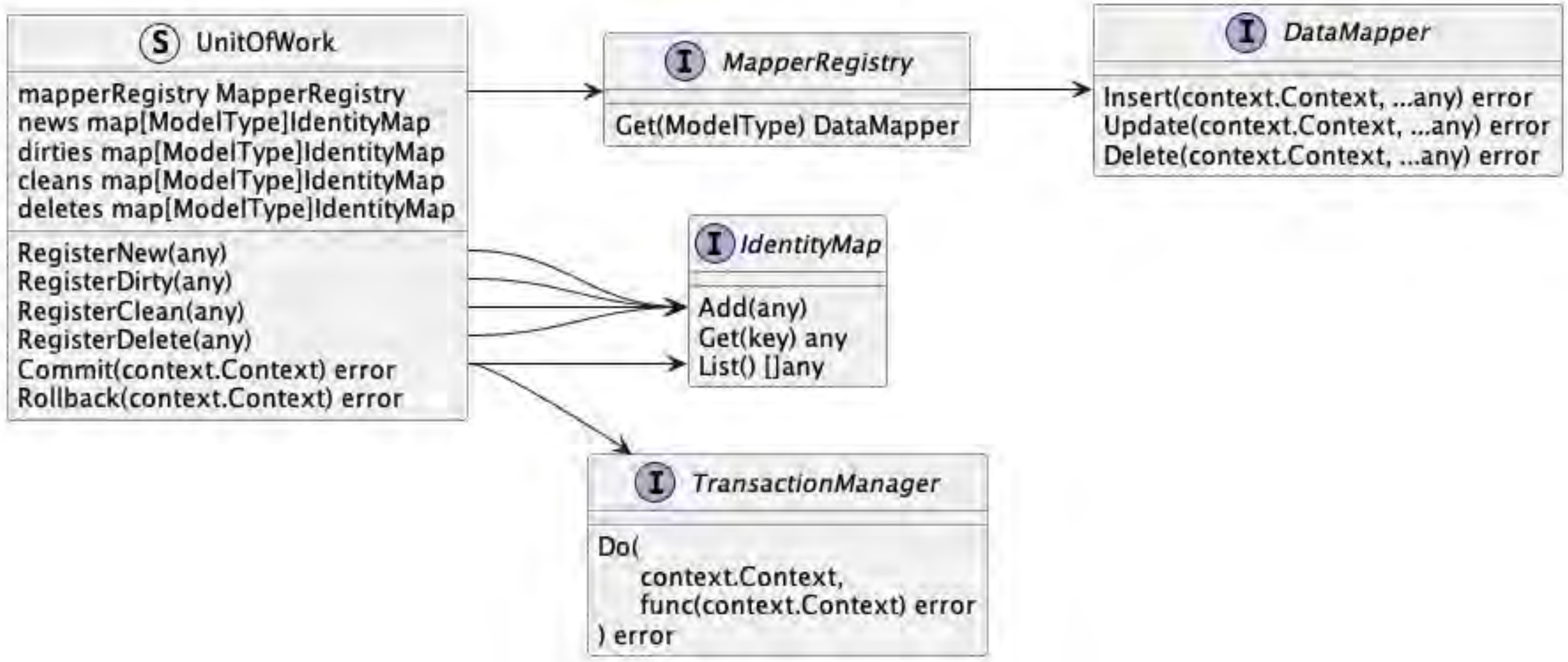
## Class Diagram of UoW


S UnitOfWork
mapperRegistry MapperRegistry
news map[ModelType]IdentityMap
dirties map[ModelType]IdentityMap
cleans map[ModelType]IdentityMap
deletes map[ModelType]IdentityMap
RegisterNew(any)
RegisterDirty(any)
RegisterClean(any)
RegisterDelete(any)
Commit(context.Context) error
Rollback(context.Context) error
I IdentityMap
Add(any)
List() []any
S DB
Begin() *Tx
I Tx
Rollback error
Commit() error

## Class Diagram of UoW


S UnitOfWork
mapperRegistry MapperRegistry
news map[ModelType]IdentityMap
dirties map[ModelType]IdentityMap
cleans map[ModelType]IdentityMap
deletes map[ModelType]IdentityMap
RegisterNew(any)
RegisterDirty(any)
RegisterClean(any)
RegisterDelete(any)
Commit(context.Context) error
Rollback(context.Context) error
I IdentityMap
Add(any)
Get(key) any
List() []any
I TransactionManager
Do(
context.Context,
func(context.Context) error
) error

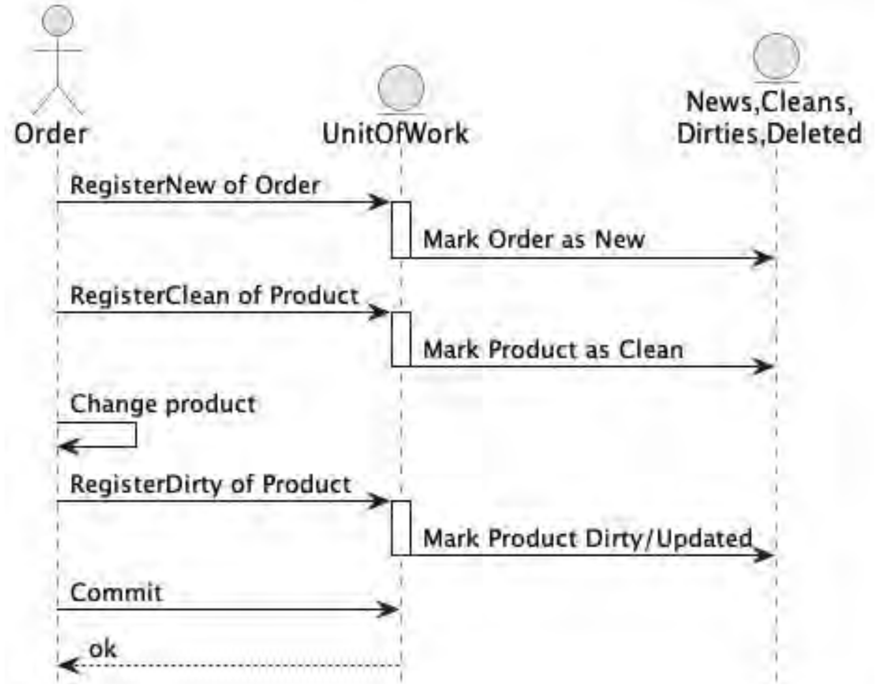
# Class Diagram of UoW


S UnitOfWork
mapperRegistry MapperRegistry
news map[ModelType]IdentityMap
dirties map[ModelType]IdentityMap
cleans map[ModelType]IdentityMap
deletes map[ModelType]IdentityMap
RegisterNew(any)
RegisterDirty(any)
RegisterClean(any)
RegisterDelete(any)
Commit(context.Context) error
Rollback(context.Context) error
I MapperRegistry
Get(ModelType) DataMapper
I DataMapper
Insert(context.Context, ...any) error
Update(context.Context, ...any) error
Delete(context.Context, ...any) error
I IdentityMap
Add(any)
Get(key) any
List() []any
I TransactionManager
Do(
context.Context,
func(context.Context) error
) error

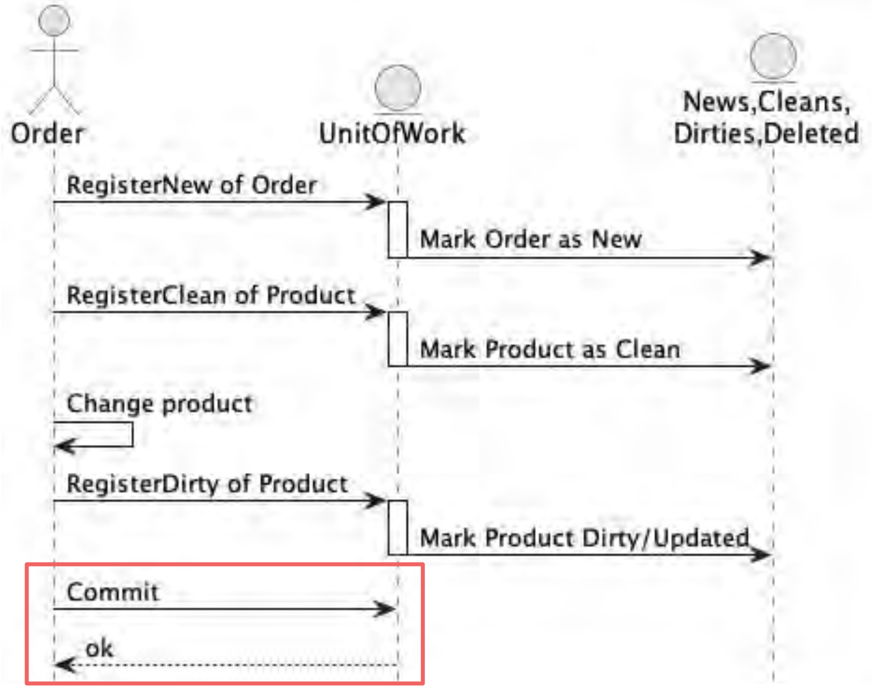
# Order Use Case

```
func Order(ctx context.Context, in In) (err error) {
   uow := NewUoW()
   defer func() {if err ≠ nil {uow.Rollback()}}()

   err = userClient.CheckExist(ctx, in.UserID)

   order, err := NewOrder(in.ProductID, in.Count)
   uow.RegisterNew(order)

   product, err := productRepo.GetByID(ctx, in.ProductID)
   uow.RegisterClean(product)

   err = product.WriteOff(in.Count)
   uow.RegisterDirty(product)

   return uow.Commit(ctx)
}
```

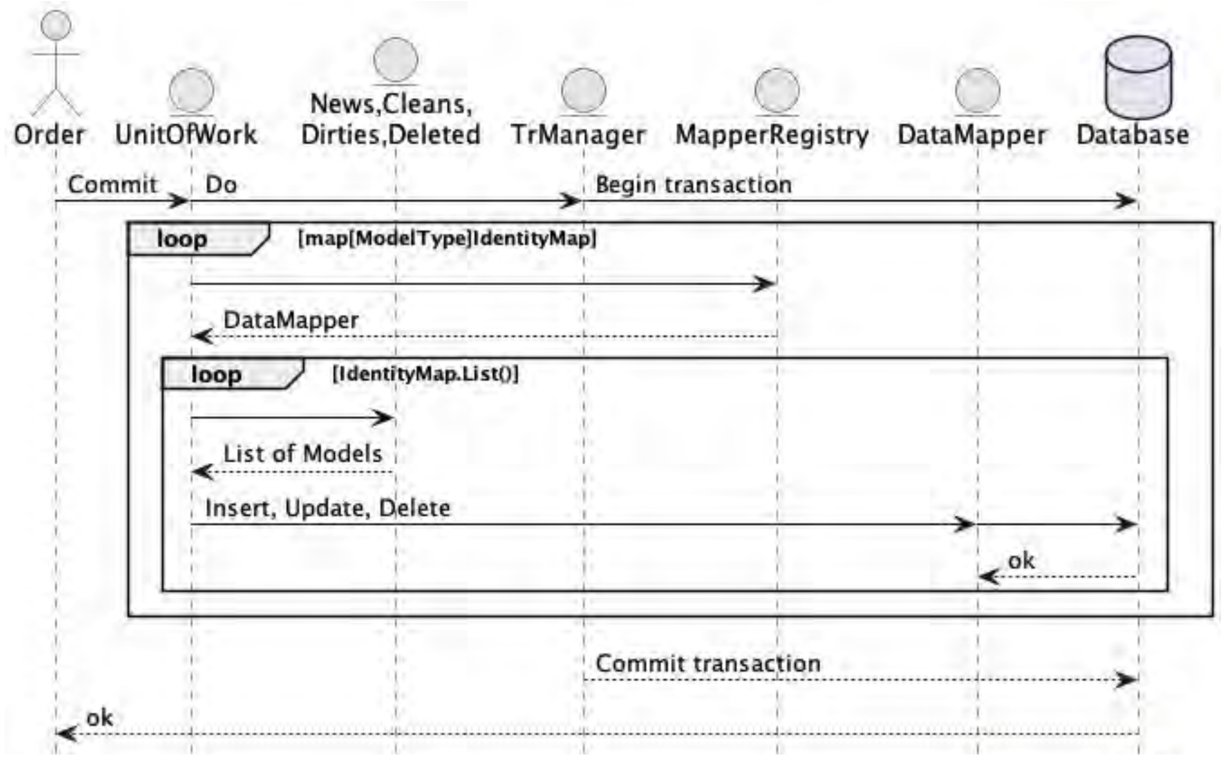
# Sequence Diagram of Use Case



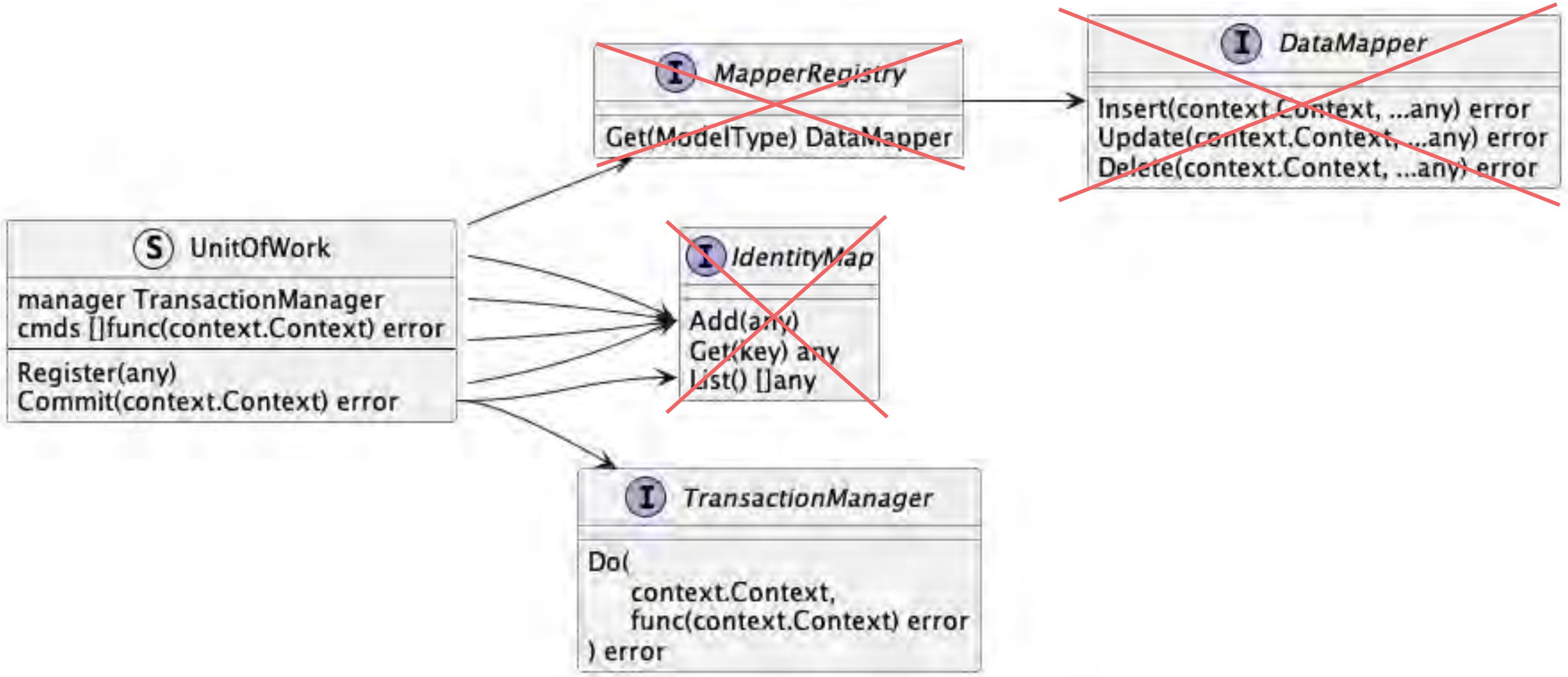
# Sequence Diagram of Use Case



# Sequence Diagram of Commit



# Simplest Variation of UoW



S UnitOfWork
manager TransactionManager
cmds []func(context.Context) error
Register(any)
Commit(context.Context) error
I MapperRegistry
Get(ModelType) DataMapper
I DataMapper
Insert(context.Context, ...any) error
Update(context.Context, ...any) error
Delete(context.Context, ...any) error
I IdentityMap
Add(any)
Get(key) any
List() []any
I TransactionManager
Do(
context.Context,
func(context.Context) error
) error

## Simplest Variation of UoW

```go
type uow struct {
    mu      sync.RWMutex
    manager Manager
    cmds    []Cmd
}

func (u *uow) Register(_ context.Context, cmd Cmd) error {
    u.mu.Lock()
    defer u.mu.Unlock()

    u.cmds = append(u.cmds, cmd)

    return nil
}
```

# Simplest Variation of UoW

```go
func (u *uow) Commit(ctx context.Context) error {
   u.mu.Lock()
   defer u.mu.Unlock()

   return u.manager.Do(ctx, func(ctx context.Context) error {
       for _, cmd := range u.cmds {
           if err := cmd(ctx); err ≠ nil {
               return err
           }
       }
       u.cmds = nil

       return nil
   })
}
```

# Simplest Variation of UoW

```go
func (u *uow) Commit(ctx context.Context) error {
   u.mu.Lock()
   defer u.mu.Unlock()

   return u.manager.Do(ctx, func(ctx context.Context) error {
       for _, cmd := range u.cmds {
           if err := cmd(ctx); err ≠ nil {
               return err
           }
       }
       u.cmds = nil

       return nil
   })
}
```

## Simplest Variation of UoW

```go
func (u *uow) Commit(ctx context.Context) error {
   u.mu.Lock()
   defer u.mu.Unlock()

   return u.manager.Do(ctx, func(ctx context.Context) error {
       queries := make([]Query, 0, len(u.cmds))
       for _, cmd := range u.cmds {
           query, err := cmd(ctx)

           queries = append(queries, query)
       }
       u.cmds = nil

       return u.dbExec.Run(queries...)
   })
}
```

## Ready Libraries?

- freerware/work

## What and When to Use

- Repository

## What and When to Use

- Repository
- Transaction manager

## What and When to Use

- Repository
- Transaction manager
- Unit Of Work

avito.tech

## Ilia Sergunin

t2m.io/conf42.trm

bit.ly/avitotrm

bit.ly/iasergunin

## Questions?
## Let’s go to the comments.