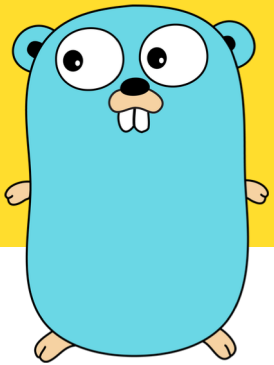
Базы данных



Tinkoff.ru

План занятия

1. Работа с реляционными БД
2. ORM
3. Логирование
4. 3-х уровневая архитектура приложения

Работа с реляционными БД

Структура БД

Пользователи

Имя	Тип	Обязательное
id	bigserial	да
name	text	да
email	text	да
password	text	да
birthday	timestamp	нет
created_at	timestamp	да
updated_at	timestamp	да

Структура БД

Роботы

Имя	Тип	Обязательное
id	bigserial	да
creator_id	bigserial	да
title	text	да
description	text	да
yield	numeric(19, 8)	да
created_at	timestamp	да
updated_at	timestamp	да

Пакет sql

https://golang.org/pkg/database/sql/

Пакет для работы с реляционными БД. Содержит все необходимые функции и методы: Open, Close, Exec, Query, QueryRow, Begin, Commit, Rollback, Prepare.

Функции и методы пакета sql

- Open – открывает соединение с БД
- Close – закрывает соединение. Если коннект не закрыть, он останется открытым на стороне БД
- Exec – выполняет запрос, возвращает интерфейс `sql.Result` (используется для DELETE, UPDATE, TRUNCATE, …)

Пакет sql

- Query – выполняет запрос и возвращает курсор `*sql.Rows`
- QueryRow – выполняет запрос, умеет вычитывать один результат, возвращает `*sql.Row`
- Begin – открывает транзакцию, которую обязательно нужно либо завершить (Commit), либо откатить/отменить (Rollback)

Подключение к БД

```
1  dsn := "postgres://user:passwd@localhost:5432/fintech" +
2     "?sslmode=disable&fallback_application_name=fintech-app"
3  db, err := sql.Open("postgres", dsn)
4  if err != nil {
5     log.Fatalf("can't connect to db: %s", err)
6  }
7
8  if err = db.Ping(); err != nil {
9     log.Fatalf("can't ping db: %s", err)
10 }
11
```

```
2019/03/05 21:12:59 can't connect to db: sql: unknown driver "postgres"
(forgotten import?)
```

Подключение драйвера

```
1  import _ "github.com/lib/pq"
2
3  func main() {
4     dsn := "postgres://user:passwd@localhost:5432/fintech" +
5        "?sslmode=disable&fallback_application_name=fintech-app"
6     db, err := sql.Open("postgres", dsn)
7     if err != nil {
8        log.Fatalf("can't connect to db: %s", err)
9     }
10
11    if err = db.Ping(); err != nil {
12       log.Fatalf("can't ping db: %s", err)
13    }
14    ...
15 }
16
```

Получение списка всех пользователей – 1

```
1  const selectUsersQuery = `SELECT id, name, email, password, birthday,
   created_at, updated_at FROM users ORDER BY id`
2  rows, err := db.Query(selectUsersQuery)
3  if err != nil {
4     return nil, fmt.Errorf("can't exec query to get users: %s", err)
5  }
6  defer rows.Close()
7
8  // Теперь работаем с объектом *sql.Rows
9
```

Получение списка всех пользователей – 2

```
10 var users []User
11 for rows.Next() {
12    var u User
13    err = rows.Scan(&u.ID, &u.Name, &u.Email, &u.Password, &u.Birthday,
14       &u.CreatedAt, &u.UpdatedAt,
15    )
16    if err != nil {
17       return nil, fmt.Errorf("can't scan row: %s", err)
18    }
19
20    users = append(users, u)
21 }
22
23 if err = rows.Err(); err != nil {
24    return nil, fmt.Errorf("rows return error: %s", err)
25 }
26
```

Смотрим код

(Query, QueryRow, Prepare, Scanner, Storage)

Использование scanner-а

```
1  type sqlScanner interface {
2     Scan(dest ...interface{}) error
3  }
4
5  const userFields = `id, name, email, password, birthday, ` +
6     `created_at, updated_at`
7
8  func scanUser(scanner sqlScanner, u *User) error {
9     return scanner.Scan(&u.ID, &u.Name, &u.Email, &u.Password,
10 &u.Birthday,
11        &u.CreatedAt, &u.UpdatedAt,
12    )
13 }
14
```

Паттерн Repository (storage)

```
1  type UsersStorage struct {
2     statementStorage
3     findByIDStmt *sql.Stmt
4  }
5  const findUserByIDQuery = `SELECT ` + userFields + ` FROM users WHERE id
   = $1`
6
7  func (s *UsersStorage) FindByID(id int64) (*User, error) {
8     var u User
9     row := s.findByIDStmt.QueryRow(id)
10    if err := scanUser(row, &u); err != nil {
11       return nil, errors.Wrapf(err, "can't scan user by id %d", id)
12    }
13
14    return &u, nil
15 }
16
```

Query builders

Если запросы сложные, то можно использовать query builder (например, goqu https://github.com/doug-martin/goqu)

```
qb := s.goqu.
    From(goqu.I("feed")).
    Prepared(true).
    Select(goqu.L(feedFields)).
    Where(goqu.I("published_at").Lt(publishedAt)).
    Order(goqu.I("published_at").Desc()).
    Limit(filter.Limit)

if len(filter.ContentTypes) > 0 {
    qb = qb.Where(goqu.I("content_type").In(filter.ContentTypes))
}

```

Преимущества работы с sql

- Простой и понятный код
- Контролируемость запросов
- Простой переход между реляционными БД

Недостатки работы с sql

- Boilerplate code (шаблонный, однотипный)
- Легко опечататься
- Валидация запросов на уровне базы
- Отсутствие типизации

ORM

Impedance mismatch

Это набор концептуальных и технических трудностей, которые часто встречаются, когда реляционная СУБД обслуживается прикладной программой, написанной в объектно-ориентированном стиле.

Основная сложность – несоответствие объектов в коде с со структурой таблиц в БД.

Impedance mismatch

```
type Lot struct {
    ID      int64
    Title   string
    Creator *User
    ...
}
```

Структура робота содержит создателя как указатель на структуру `User`.

Как хранить такой объект?

А если у робота есть массив тегов, как с ними работать?

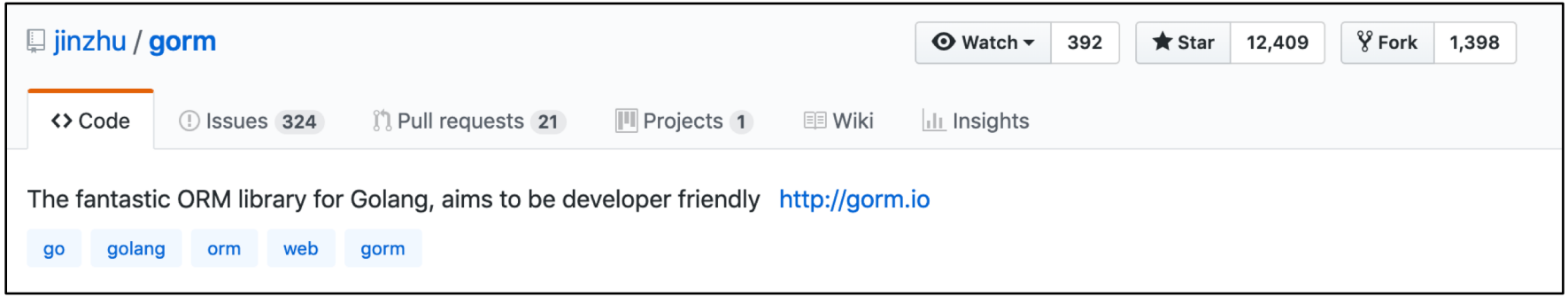
ORM

Решение – ORM

ORM (Object-Relational Mapping) – **технология** программирования, которая связывает базы данных с концепциями объектно-ориентированного программирования.

Часто под ORM подразумевают библиотеку для работы БД, осуществляющую объектно-реляционное отображение.

gorm



Наиболее популярная ORM – gorm.

Работает с тегами, есть встроенные миграции

Подключение к БД с gorm

```
1  import _ "github.com/jinzhu/gorm/dialects/postgres"
2
3  func main() {
4     dsn := "postgres://user:passwd@localhost:5432/fintech" +
5        "?sslmode=disable&fallback_application_name=fintech-app"
6     db, err := gorm.Open("postgres", dsn)
7     if err != nil {
8        log.Fatalf("can't connect to db: %s", err)
9     }
10    defer db.Close()
11    ...
12 }
13
```

Добавление пользователя

```
u := User{
    Name:     ”Ivan",
    Email:    ”ivan.ivanov@tinkoff.ru",
    Password: "password",
}

db = db.Create(&u)
if err := db.Error; err != nil {
    return errors.Wrap(err, "can't create user")
}

```

Получение пользователя по email

```go
var u User
const email = "ivan.ivanov@tinkoff.ru"
db = db.Where(&User{Email: email}).First(&u)
if err := db.Error; err != nil {
    return errors.Wrapf(err, "can't find user by email %q", email)
}

```

Сгенерируется запрос

```
SELECT * FROM "users"  WHERE ("users"."email" = 'ivan.ivanov@tinkoff.ru')
ORDER BY "users"."id" ASC LIMIT 1
```

gorm

С gorm можно получать более сложные объекты (например, связанные с foreign key). Все возможности описаны в документации.

Основная мысль: gorm (как и любая orm) инкапсулирует работу с SQL, предлагая использовать объектный подход.

Преимущества ORM (в общей случае)

- Работа с объектами
- Изменения схемы БД отражены в структурах, минимум изменений
- Отсутствие boilerplate code
- Наличие типизации (orm на основе кодогенерации)

Недостатки ORM

- Неконтролируемые запросы
- Зависимость от внешних библиотек (появляются более новые/продвинутые, существует консервативный взгляд)
- Медленные (работающие на основе рефлексии + вносят дополнительные расходы, т.к. все на основе пакета `sql`)

Из-за своих недостатков неприменимы в highload-проектах.

Логирование

Логгирование

Зачем логировать?

1) Поиск причины ошибки
2) Исследование события (инцидента)

Что логировать?

Всё, что поможет разобраться в событии / инциденте / ошибке

Никогда так не делайте

```
1  // 1
2  err := doSomething()
3  if err != nil {
4     logger.Errorf("can't do something")
5     return err
6  }
7
8  // 2
9  logger.Debugf("open connection")
10
```

Проблема одна – бесполезное логирование.

1) Логированием ошибки занимается тот участок кода, который подавляет ошибку
2) При большом количестве записей – бесполезно

Количество записей

Количество записей в лог-файле должно быть минимальное необходимое для разбирательства в инциденте.

Много записей в лог-файле – плохо, т.к. дополнительная нагрузка (CPU, storage).

Мало записей – плохо, т.к. сложно разобраться в инциденте.

Отсутствие записей в лог файле – вы слепы.

Логгеры

1) `fmt.Printf`
2) Пакет `log`
3) Logrus, https://github.com/sirupsen/logrus
4) Zap, https://github.com/uber-go/zap

Требования к логгеру

Основные:

1) Строгий, единообразный формат
2) Разные уровни логирования

Дополнительные:

1) Разные форматы (json, console)
2) Поля (fields)

Оформление логгера

Часто над библиотекой с логгером делают обёртку, которую располагают в `pkg/log`. Это делается, чтобы конфигурировать логгер в одном месте, при случае с легко заменить его, делать обёртки над логгером (кастомные функции в пакете log).

Контекстное логирование

Проблема: в логе видим ошибку, но не можем определить другие записи в лог-файле, связанные с этой ошибкой.

Решение: добавить каждому запросу идентификатор, по которому можно будет сгруппировать все записи в лог-файле.

Общепринятого названия нет, обычно `request_id`, `tracking_id`, etc.

Реализация на основе graylog

```
1  // package log
2
3  type ctxKey int
4
5  const fieldsKey ctxKey = 0
6
7  func getFields(ctx context.Context) (Fields, bool) {
8     fields, ok := ctx.Value(fieldsKey).(Fields)
9     return fields, ok
10 }
11
```

Реализация на основе graylog

```
12 // package log
13 func WithFields(ctx context.Context, fields Fields) context.Context {
14    if ctx != nil {
15       if f, ok := getFields(ctx); ok {
16          for k, v := range fields {
17             f[k] = v
18          }
19          return context.WithValue(ctx, fieldsKey, f)
20       }
21    }
22
23    return context.WithValue(ctx, fieldsKey, fields)
24 }
25
```

Реализация на основе graylog

```
26 // package log
27 func Log(ctx context.Context, logger Logger) Logger {
28     if ctx != nil {
29         if f, ok := getFields(ctx); ok {
30             return logger.WithFields(f)
31         }
32     }
33     return logger
34 }
35
```

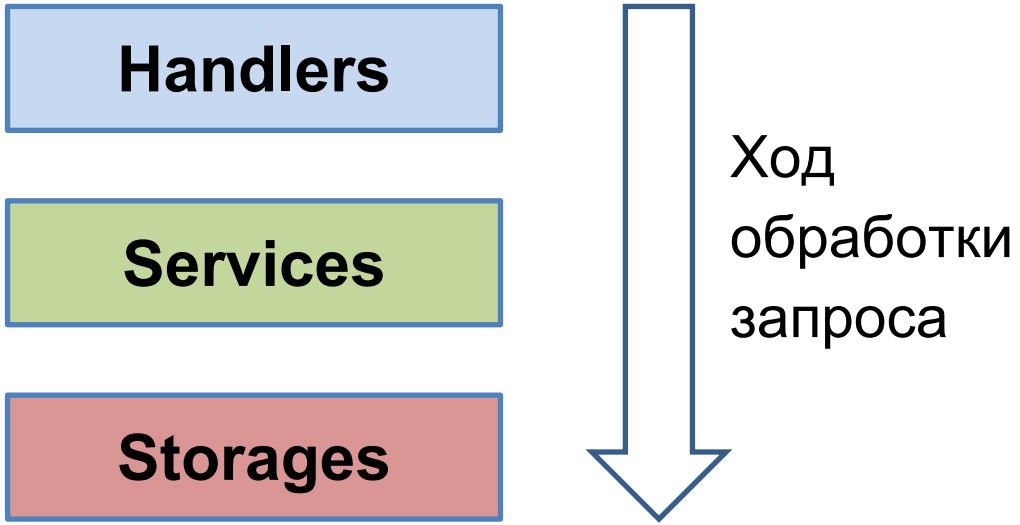
Реализация на основе graylog

```
1  package middleware
2
3  func TrackingID(next http.Handler) http.Handler {
4     fn := func(w http.ResponseWriter, r *http.Request) {
5        trackingID := generateTinyTrackingID()
6        ctx := log.WithFields(r.Context(), log.Fields{"tracking_id":
7  trackingID})
8        ctx = context.WithValue(ctx, TrackingIDKey, trackingID)
9        next.ServeHTTP(w, r.WithContext(ctx))
10    }
11    return http.HandlerFunc(fn)
12 }
13
```

3-х уровневая архитектура

3-х уровневая архитектура приложения

Некий аналог MVC. Суть: каждый слой выполняет определённый ряд действий.



3-х уровневая архитектура приложения

Handlers – занимаются обработкой входящих запросов (например, http); преобразует входящий формат в объектный формат бизнес-логики.

Services – реализуют бизнес логику.

Storages – работают с базой данных, представляют собой ORM.

Обратная связь

Спасибо за внимание

Tinkoff.ru